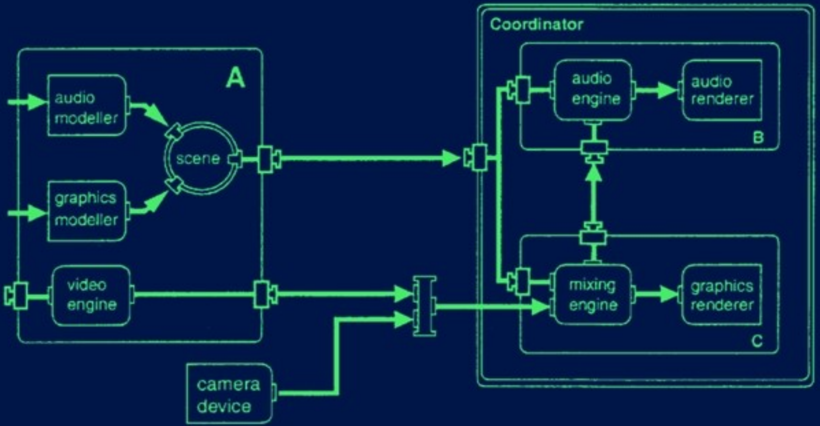
LNCS Tutorial

David J. Duke   Ivan Herman
M. Scott Marshall

# PREMO: A Framework for Multimedia Middleware

Specification, Rationale, and Java Binding



Springer

Lecture Notes in Computer Science 1591
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
*Berlin*
*Heidelberg*
*New York*
*Barcelona*
*Hong Kong*
*London*
*Milan*
*Paris*
*Singapore*
*Tokyo*

David J. Duke   Ivan Herman   M. Scott Marshall

# PREMO: A Framework for Multimedia Middleware

Specification, Rationale, and Java Binding

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Authors

David J. Duke
The University of York, Department of Computer Science
Heslington, York YO1 5DD, UK
E-mail: duke@cs.york.ac.uk

Ivan Herman
M. Scott Marshall
Centrum voor Wiskunde en Informatica (CWI)
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
E-mail: {ivan.herman; scott.marshall}@cwi.nl

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

PREMO: a framework for multimedia middleware : specification, rationale, and java binding / David J. Duke . . . (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1591)
ISBN 3-540-66720-2

CR Subject Classification (1998): H.5.1, D.2, D.4, H.5, I.3, C.2, D.1.5, D.1.3

ISSN 0302-9743
ISBN 3-540-66720-2 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN: 10704630 06/3142 - 5 4 3 2 1 0 Printed on acid-free paper

# Preface

Early in 1998, SC24, the subcommittee of ISO/IEC JTC 1 concerned with computer graphics and image processing, completed work on a new standard for multimedia presentation, called PREMO (PResentation Environment for Multimedia Objects), and published under the official reference ISO/IEC 14478. The original proposal for PREMO was for a new computer graphics standard, to be based explicitly on an object-oriented approach. Such an approach was seen as timely, given that object-oriented design and programming had rapidly become established, and work on a number of object-oriented APIs for computer graphics had generated interest within the graphics community for this technology (Inventor, the precursor of OpenInventor, is probably the best known example). Development of a new standard was also seen as an opportunity to address further technological issues. First, the new standard should encompass *other* media, such as video, audio (both captured and synthetic), and in principle be extensible to new modalities such as haptic output and speech or gestural input, which have become increasingly integrated within graphics applications; virtual environments and systems for visualization being prime examples. The second requirement was that the standard should allow the construction of distributed systems, where parts of a system involved in the generation, processing, or the presentation of media data could be distributed across geographically remote sites, interacting through a network.

Although the original goals for the development of PREMO included the detailed specification of an API for multimedia programming, including all kinds of rendering and media-coding facilities, it soon became clear that such goals were unrealistic. The diversity of requirements for various applications and the wide range of different techniques made the development of a detailed specification problematic. Instead, *interoperability* became the key issue: *existing* tools, applications, and programming interfaces should be able to cooperate, even if they come from different implementations and vendors. The term "middleware" came to the fore, denoting a software layer between the operating system facilities and application programs: in this way PREMO evolved into a middleware specification for multimedia programming.

Although PREMO defines objects for implementing multimedia middleware, the emphasis on interoperability means that PREMO also functions as a *reference model* for distributed multimedia. Concepts common to a range of approaches in this area have been described and integrated in the PREMO model, and consequently the standard has an important role in education, and in promoting cooperation between programmers involved in multimedia development projects across potentially heterogeneous platforms.

The text of an International Standard is usually dry, and notoriously difficult to read. Although this book does not replace the official text, its goal is to provide a more readable version of the concepts, to present some of the more interesting details of the PREMO multimedia objects, to highlight the reasons for specific design decisions, and to give simple examples which clarify the underlying concepts. If the goal of the reader is to implement the PREMO standard, this book should aid in understanding the precise specification of the ISO text. However, the book should also be useful for students and professionals whose goal is to gain a better understanding of the issues involved in distributed multimedia, regardless of the intricate details of the PREMO standard; this group probably represents the majority of our readers.

Obviously, PREMO is the result of team work, which involved experts from four continents and more than 10 countries. It is impossible to list all the people who, for a shorter or a longer period, participated in the work. Nevertheless, we would like to mention the contributions of three people who played particularly important roles. Horst Stenzel (FH Köln, Germany) was the rapporteur of the working group within ISO which was responsible for the development of PREMO. It was his task to coordinate the development of the standard. James Van Loo (Sun Microsystems Inc., USA) was a co–editor of the document, and was instrumental in integrating the so–called Multimedia Systems Services definition, which became the core of the final PREMO document. Finally, David Duce (Rutherford Appleton Laboratory, UK) coordinated the ERCIM Computer Graphics Network which, between 1993 and 1997, played a seminal role in the precise specification of large portions of the standard. We express our gratitude to them, as well as to all experts who participated in the development of PREMO; this book is the result of their work.

February 1999

David Duke
Ivan Herman
Scott Marshall

# Contents

# Chapter 1

# PREMO: A Standard for Distributed Multimedia

## 1.1 Introduction

The use of multimedia is now so widespread that the term ‘multimedia computing’ has become almost redundant. Few people today would conceive of purchasing or using a computer system that was *not* capable of displaying or processing multimedia data. Standards are now available for the encoding, transport and presentation of a rich variety of media data. Many of these, such as JPEG, MPEG [55], MIDI and VRML [54] are well known even among non-professional computer users. New standards, such as PNG [94] and SMIL [95] are under development in response to the opportunities and needs created by the World Wide Web. This apparent wealth of media standards makes it all the more important to situate PREMO and understand its role:

- *PREMO is a presentation environment.* PREMO, like previous SC24 standards, aims at providing a standard “programming” environment in a very general sense. The aim is to offer a standardised, hence conceptually portable, development environment that helps to promote portable multimedia applications. PREMO concentrates on the application program interface to “presentation techniques”. This is primarily what differentiates it from other multimedia standardization projects.
- *PREMO is intended for multimedia presentation.* Whereas earlier SC24 standards concentrated either on synthetic graphics or image processing systems, multimedia is considered here in a very general sense. High–level virtual reality environments, which mix real–time 3D rendering techniques with sound, video, or even tactile feedback, and their effects, are also within the scope of PREMO.

In the remainder of this section, we will explore these two points in more detail and establish the fundamental rationale for the technical content and approach that is contained in the remainder of this book.

### 1.1.1 What PREMO *Is*

Programming interfaces for graphics (“graphics packages”) are now widely known and used. These include *de jure* standards developed within ISO, specifically GKS [48] and PHIGS [5], and industry-developed platforms such as OpenGL [93] and Inventor [89] that have now become *de facto* standards themselves. The process is ongoing, with a new generation of graphics applications emerging based on the Java technologies (e.g. Java3D), and also in response to the needs and opportunities of web-based applications (e.g. VRML). In contrast, programming interfaces for multimedia are rather less well known; while toolkits for multimedia applications have been developed, for example MET++ [1] and MADE [42], standards for multimedia have concentrated largely on formats for the storage and transport of media, declarative models of media content (for example HyTime [68]). While the interface to presentation engines for such formats does provide a starting point for the applications programmer, the level of control over media processing that these affords is significantly lower than can be achieved in computer graphics. And significantly, none of the existing presentation models or engines integrates their media content with synthetic graphics.

The distinction between synthetic graphics and other presentation media may reflect the different communities in which the fundamental developments took place (e.g. much of the early interest in multimedia was stimulated by applications in publishing and human-computer interaction, whereas graphics originally had stronger links with engineering and scientific applications). Irrespective of these differences in origin, two technological trends have meant that there is now a growing need to integrate these two threads of activity. At one end of the cost-performance spectrum, the virtual environments and visualisation are emerging as mature technologies with needs that encompass both synthetic graphics and other media, e.g. 3D audio, acoustic and haptic rendering. At the other end of the cost spectrum, the availability of powerful, low-cost personal computing platforms has made it feasible to develop multimedia applications for mass markets, and for users of such machines to experiment with multimedia. An issue that spans this spectrum of applications is how application programs can access, construct, and control multimedia and graphics presentation. This is the context in which PREMO has been designed.

## PREMO as Middleware

The term "middleware" has come to the fore in recent years. It refers to a software layer, between the fundamental services of an operating system and more specific application development environments. PREMO provides a level of middleware which supports the implementation of a range of processing models for multimedia presentation. As a form of middleware, PREMO does not define stand-alone services in the way, for example, that a GKS renderer does. Instead, it provides an environment where various, vendor–specific components can cooperate. The middleware nature of PREMO has implications for how the software objects defined by the Standard are described. On the one hand, these must not be too detailed, otherwise it would restrict the range of possible implementations, but on the other hand these objects must provide a non-trivial set of services. This strive for balance has fundamentally shaped the Standard.

Why is middleware important? Consider, for example, the task of implementing a distributed multimedia application such as a multi-platform video–conferencing system. Due to the variety of available media formats, resource requirements, means of distribution control, etc., a significant portion of such an application is dedicated to issues like configurability, adaptability, access to remote resources, distribution, etc. A similar level of adaptiveness is also required when using media in combination as with, for example synthetic graphics, video, and computer animation. No one applications package addresses such a variety of needs, and without middleware such as PREMO, much of this infrastructure has to be developed from scratch or adapted from a similar application. And the costs involved in modifying software to meet new demands are well known.

In addition to enabling interoperation, the existence of middleware such as PREMO can also assist in system evolution. The variety of graphics formats, available primitives, animation algorithms, etc., continues to expand, and portable applications increasingly have to adapt to an evolving environment. PREMO assists in this process by factoring out at least some of the technological constraints into components that can be interchanged and replaced, and by providing a flexible and extensible architecture in which new software components can be defined for use by existing applications.

Multimedia presentation is not the only concern that is open to support by middleware. Another, well known, example is architectural support for distributed object–oriented applications, as is provided by OMG's CORBA[71].[1] Although PREMO itself is not related to the OMG specifications, PREMO should be viewed as a multimedia–oriented extension of the basic object services and architecture provided by systems like OMG or, as another example, Java's RMI services [39]. Seen in this way, PREMO fills the gap between the application-independent set of facilities offered by OMG, and a distributed multimedia application. Indeed, the relationship between PREMO and a distributed object–oriented architecture is so close, it would be ill-advised to attempt an implementation of PREMO without the use of such services. Exactly how the PREMO specification builds on the concept of distributed multimedia *without* committing to a particular model will be seen in Chapter 6.

## PREMO as a Reference Model

Because PREMO describes an implementation environment (a prototype is currently under preparation in Java, see Chapter 4), the specification encompasses a range of concepts needed in multimedia systems development. By providing a broad, application independent model of media processing, the specification itself also serves as a reference model for distributed multimedia. This is significant, as in practice, "portability of programmers" is almost as important as the "portability of programs". Although only the latter role of information processing standards is usually publicised by organizations like ISO, the need for "programmer portability" in this area was also considered to be a major goal for PREMO. Having a common, well understood set of principles and techniques as a reference point greatly helps in understanding both the specificities and the commonalities of various multimedia programming environments. To achieve this goal, the PREMO specification deliberately sets out a number of details which are sometimes hidden in other systems. As a reference model, PREMO is not only significant in a didactical sense; a unifying set of concepts may play an important role in classifying, relating and organising the growing range of software toolkits that are available to the

[1] In fact, a liaison existed between OMG and the relevant ISO group, during the development of PREMO, which clearly influenced the design of the Standard.

potential developer of a multimedia system. Without such concepts, this technological cornucopia is in danger of becoming an anarchic ensemble of incompatible and incomparable artifacts.

### 1.1.2 What PREMO *Isn't*

The characteristics that define what PREMO is – middleware and reference model – also reflect what PREMO is not. In particular, PREMO is intended to build on and utilise existing media standards, not to replace them. Given that there are standards in place for media formats and processing, these are two concerns that PREMO does not address.

#### PREMO Is *Not* a Media Format

The PREMO specification does not describe any new format for the representation and storage of media data. Instead, the Standard makes it quite clear that the data processed by PREMO-based applications is expected to be stored in existing formats; ALAW, JPEG, MHEG, MIDI, MPEG [55], SMIL [95], VRML [54], to name a few. What PREMO does provide are mechanisms by which new PREMO objects can be defined for new formats, and by which existing objects can coordinate the formats that they use to exchange and process media data.

#### PREMO Is *Not* a Media Engine

The object types defined in the PREMO standard are not sufficient in themselves to realise a working multimedia application. To do this would have required the Standard to commit to particular kinds of media processors and renderers, with specific interfaces. All that this would achieve would be to add yet another type of media engine into the growing collection of such devices. Instead, PREMO provides a number of object types that can act as "wrappers" around existing engines, and allow these to be used within a processing network involving other devices that may be based on quite different media formats or models. Rather than thinking of PREMO as a media engine, a somewhat better analogy is to view PREMO as a software architecture for multimedia applications; the objects defined by PREMO represent the basic constructs, the building blocks, for multimedia applications. Even this analogy is not quite the whole story though. Although parts of the PREMO specification provide building blocks that are "shaped" for supporting a particular architectural model of an application, these in turn rely on a set of lower-level PREMO objects, and users of PREMO is free to build on these, or modify the higher-level components, in order to instantiate whatever model of multimedia architecture that is most appropriate for their needs.

Just as PREMO is not a media engine, it is not a complete environment, either. It does not, for example, provide a framework for quality of service management. This may seem strange, since quality of service is a particularly fundamental problem with multimedia applications. However, quality of service management is currently bound tightly up with network management issues; as of yet, there is no emerging consensus on what application mechanisms are needed to implement quality of service, and indeed, it seems probably that, like the concept of a network, 'quality of service' actually spans a whole range of levels of concern, from well known physical properties such as bandwidth and latency of raw transmissions, up to questions that impinge on the eventual presentation of the data, for example synchronization constraints between lip motion in video frames and the corresponding speech in an audio stream. What PREMO does provide here is a basic set of hooks and facilities which a quality of service management protocol is free to utilise for monitoring and realising its requirements.

### PREMO Is *Not* a User-Oriented Specification

In addition to the technical problems of building a media application, multimedia systems designers need to address the question of how well a particular media system (both in terms of technology, and media content) meets the demands of its users. Like the issue of quality of service, usability involves a spectrum of concerns, from low level issues of signal quality, through questions about the cognitive resources and processes needed to interact with an application, through to questions of how a particular system is situated in the work context and environment of its users. These human factors must obviously be addressed by media systems designers by making appropriate use of the technologies at their disposal. PREMO is one such technology – the specification itself does not describe how it should be used to realise user requirements.

## 1.2 Formal Description Techniques and PREMO

Formal description techniques (FDTs) are mathematically–based notations that can be used to describe the behaviour of software and hardware systems. They offer a highly expressive language for characterising what a component of some system should do, without the need to describe explicitly the algorithm or mechanism needed to bring about that behaviour. Various formal description techniques are in use — LOTOS [46] and Z [78] are good examples of the range of possibilities. They differ in the mathematical structures that underpin the notation, and the different approaches to specification that emerge from the mathematics offer different trade–offs, for example between expressiveness and support for automated analysis.

ISO has investigated the use of formal description techniques in the development of International Standards, and has developed a three–phase model for the adoption of formal methods in the Standardization process (see Section 10.4 of [56]). These encompass the use of a formal description by the committee developing the Standard (phase one), through to the third phase in which the formal description, accompanied by a natural language commentary, defines the provisions of the Standard. Experience with and acceptance of formal methods means that the third phase is simply not practical at present. However, serveral members of the PREMO rapporteur group had successfully used formal description methods in the context of other graphics standards, and in July 1993, SC24 appointed a Special Rapporteur for Formal Description Languages

(G.J. Reynolds, then at CWI, Amsterdam), and invited him to provide an initial report on the applicability of formal description techniques to SC24 standards [72]. This report recommended that the formal description technique “Object–Z” [24,25] be used in the development of PREMO, as a way of gaining further insight into the design of the object types needed in the Standard.

Subsequent work with formal methods and PREMO concentrated on two areas that were proving a source of difficulties within the committee work, specifically the PREMO object model, and the general facilities for synchronization. Work on the object model has been published as a paper [19]. The initial work on synchronization has been published as a technical report [20], while subsequent work exploring behavioural aspects of synchronization is described in [26]. We will not make use of the formal description of PREMO in the book, but the interested reader may consult the references above, or a recent paper summarising this aspect of the development [23].

## 1.3 Structure of the Book

The main objective of this book is to present a detailed description of the PREMO standard. From the background material in section 1.1, it should already be clear that PREMO encompasses a range of concerns, spanning levels of multimedia presentation from low level synchronization of media streams through to high level facilities for interoperation and coordination of devices. This complexity presents something of a challenge when attempting to provide a coherent account of the Standard; by starting with the low-level facilities, one risks describing a large number of detailed facilities but with little direction or motivation, while in starting with the higher-level facilities, one is forced to introduce concepts with the proviso that “these will be explained in detail later”. We have sought to avoid both of these pitfalls by giving an initial, comparatively informal description of the Standard as a self contained chapter, and then entering into the details, matching our explanation with the actual structure of the Standard.

As PREMO is intended to be independent of any particular system or programming language, the description of the official standard was deliberately written using a non-programming notation, based on the Object-Z specification language [24][25]. We have chosen not to reuse this notation here, for two reasons. It is awkward to typeset, even within a poerful typesetting language like LaTeX (which has not been used for this book). More importantly, a second objective of this book is to describe how the PREMO standard could be implemented. This requires a programming notation (along with a supporting environment for distributed objects). For this we have chosen the Java technologies of Sun Microsystems Inc.; specifically the language [36] and the RMI library [39] for remote access. And rather than use two notations for describing PREMO, we have decided to use Java throughout, both as a means of documenting what the Standard provides, and as the vehicle for discussing implementation issues. As we will see, the match between what the PREMO standard describes and what Java provides is not exact, and consequently we have dedicated an early chapter of the book to describing how we are using Java to describe PREMO. Thus, following this introduction, we have:

- *Chapter 2, An overview of PREMO*. The high level organisation of PREMO into four distinct components is explained, and subsequently the role and principle features of each component is described. This is an important chapter as it provides the underlying rationale for some of the design decisions that are described in more detail in later chapters.
- *Chapter 3, Fundamentals of PREMO*. In order to understand PREMO, it is necessary to understand the object model on which the whole PREMO approach is based. This chapter describes the first part of the Standard, which sets out PREMO's view of object orientation, other kinds of entity in a PREMO system, and assumptions about the environment of a PREMO application.
- *Chapter 4, General implementation issues*. The description of the PREMO object model is by necessity independent of any language and implementation environment. In the remainder of the book, we will be using Java to describe the object types and services defined in the Standard. The general issues that arise from using Java as the language to describe the Standard are described in the chapter. It covers aspects such as naming conventions, and the facilities beyond the core language that are needed to define PREMO.

The three chapters that follow present the content of the remaining three parts of the Standard, with each chapter addressing a particular component of PREMO.

- *Chapter 5, The foundation component*. This chapter explores the low level services and objects that underpin many approaches to multimedia presentation.
- *Chapter 6*, *Multimedia systems services component*. Higher-level abstractions over media processing devices and other resources are introduced, and their role in supporting a particular paradigm for multimedia systems is discussed.
- *Chapter 7*, *Modelling, Rendering, and interaction component*. Specialised devices for media presentation, in particular support for synthetic graphics, is introduced, along with a framework for declarative modelling of media presentations.

In conjunction with this book, a number of core object types from the Standard have been developed and placed in the public domain[1]. It is not a complete implementation of the Standard, but the most significant object types described in this text have been realised, in particular those that present interesting challenges to an implementer. This activity has been carried out, not so much with the intention of using it for subsequent systems development (though that is certainly possible), but rather to demonstrate how the requirements defined for PREMO can be met in practice.

- *Chapter 8, Detailed Java specifications of the PREMO objects*, gives the complete specification of the PREMO standard as a set of Java interfaces.
- *Appendix A, Selected implementation issues*. A number of the programming tasks facing a PREMO implementer are relatively routine; concepts such as state machines, property lists etc. are well known and should require no design hints.

[1] ftp://ftp.cwi.nl/pub/premo

However, a few requirements defined for PREMO are not trivial, and in the appendix we describe a strategy for dealing with issues such as operation dispatch modes and the creation of connections.

An index is provided at the end of the book.

## 1.4 Typographical Conventions

We have endeavoured to keep typographical conventions to a minimum. The text of the book is written in times font (thus), with italics & bold for occasional emphasis. When mentioned within the text, the names of PREMO object types (Java classes and interfaces) appear capitalised, and in italics. The same convention is used for the names of PREMO operations (Java methods) in the running text. Definitions of Java classes, packages, methods, and related code fragments, are presented as indented blocks, using `courier` font.

## 1.5 Graphical Conventions

Although UML [30] is approaching the status of de facto standard for the documentation of object-oriented systems, the notation goes beyond what is needed in most parts of this book, and we have instead opted for a simple way of documenting object-oriented structures based on the conventions used in the book "Java in a Nutshell" [28]. These are:

- class names are written in normal times font;
- interface names are written in *times italic*;
- if class (or interface) B extends class (or interface) A, B is written below A, and they are joined by a solid line; and
- if class B implements interface A, B is again drawn below A, and they are joined by a dashed line.

Shaded ovals have been used to show the grouping of classes and interfaces into packages.

# Chapter 2

# An Overview of PREMO

## 2.1 Introduction

In the introduction we established the need for a standard to address distributed multimedia and the rationale for designing the standard to be extensible. This chapter is intended to provide an overview of PREMO, and in doing so, one of the first concepts that will be addressed is the PREMO component, which was introduced into the standard to promote extensibility. Then, the four 'parts' that make up the official PREMO standard are briefly introduced. The design of each part is described in detail later in the book, together with an outline of how its key provisions can be implemented. The purpose of this chapter is to summarise the content of the components, and explain how each contributes to the overall aims of the standard.

## 2.2 The Structure of PREMO

The concepts of modularity, data abstraction and component-based design are now well established within software engineering, where structures such as classes, modules and packages are used to manage the complexity of systems development by allowing the decomposition of a design into a set of parts which can be developed independently or incrementally, before being composed to form the desired system. The object-oriented basis of PREMO allows one level of structuring. However, this is relatively fine-grained, and in practice multimedia applications require *families* of objects that can be assembled to implement particular functionalities. Today, this concept is becoming widely adopted in the form of design patterns [31]. These were, however, less well known when development of PREMO began, and consequently a somewhat simpler approach was adopted to structure the standard.

PREMO is defined as a collection of *components*, each of which provides one or more *profiles*. A component defines a collection of entities, such as object and non-object types. Object types provide services (in the form of operations that can be invoked by clients), or can have a more passive role, for example as data encapsulators. Because not all of the types defined within a component are necessarily needed in a given context, PREMO components define one or more *profiles* which consist of a cluster of entities. A component can build on (extend) the profiles of other components, in the same way that a class in object-oriented programming can be defined as an extension to existing classes. The components defined in the PREMO standard are general purpose; they provide a progressively richer, and more structured model of multimedia processing. It was the intention of the designers to realise functionality which would address specific technologies, such as 3D audio and virtual reality, or specific application domains, such as medical simulation or battlefield models. The development of new components that extend some or all of the profiles defined in the standard helps to achieve this aim. The four components of the PREMO Standard are :

1. *Fundamentals*. This specifies the object model used by PREMO and the requirements that a PREMO system places on its environment. Although the PREMO object model is similar to the core model of the OMG (Object Management Group) [69], it contains features needed to address the requirements of distributed systems.
2. *Foundation*. Object and data types that are generic to multimedia applications are defined in this component, including facilities for event management, synchronization, and time.
3. *Multimedia Systems Services*. Multimedia systems typically integrate a variety of logical and physical devices. Some examples are input and output with devices such as video editors, cameras, speakers, and processing with devices such as data encoders/decoders and media synthesizers (e.g. a graphics renderer). This component of PREMO defines the infrastructure needed to set up and maintain a network of heterogeneous processing elements for media data. These facilities include mechanisms by which media processors can advertise their properties and be configured to match the needs of a network, and can then be interconnected and controlled. MSS was originally defined by the Interactive Multimedia Association[45] and subsequently adopted by SC24 and refined into a PREMO component.
4. *Modelling, Rendering and Interaction*. The MSS component defines concepts of streams and processing resources that are independent of media content. In the MRI component, these facilities are used to define generic objects for modelling and rendering data, and basic facilities for supporting interaction. To support interoperability, the component defines a hierarchy of abstract primitives for structuring multimedia presentations. These are not sufficient in themselves to build a working presentation, but provide the abstract supertypes from which a set of concrete primitives could be derived.

Each of these components is now described in more detail.

## 2.3 The PREMO Object Model

Although with the emergence of UML [30] there is now some level of consensus on a set of concepts for object-oriented modelling, at the implementation level there still remain a number of different approaches, as represented by the range of programming languages that are claimed to support object-oriented techniques. These differences vary from the fundamental, such as whether a system is class-based, or object-based (using prototypes to define the structure of objects), to finer details, such as the various levels of visibility or accessibility that can be assigned to the components of an object.

Within a development project which uses an object-oriented target language, the choice of object model is effectively made once the target language is chosen. Indeed, the precise details of the available object model may be one criteria by which the language is chosen. In the case of PREMO, however, the situation is more complicated. Like the standards that it follows (GKS [48] and PHIGS [49]), PREMO is intended to be independent of any particular programming language. Thus, just as one can obtain a C [58] binding or a FORTRAN [16] binding for GKS, it should be possible to obtain a C++ [79] or Ada'95 [7] binding for PREMO. The need to provide this flexibility raises a number of difficult technical questions, not the least being whether it should be possible to bind PREMO to a language with no explicit support for object-oriented programming (e.g. FORTRAN'77 [82]). For now, the main point is that if PREMO is to be language independent and described in an object-oriented framework, it requires the definition of some object model that defines the concepts from which the remainder of the standard can be constructed.

One of the fundamental issues that had to be decided at an early stage in the project was whether to adopt a "classical" object-oriented approach, in which objects are instances of classes that can be arranged in a hierarchy through inheritance, or opt for a more radical approach based, for example, on the use of prototypes and delegation. The former is typical of the models that underlie object-oriented design methods, and has been in widespread use in the form of languages such as Simula [9], Smalltalk [35], and C++. Prototype-based approaches have, in contrast, been largely the concern of the research community; there has already been discussion on the value of such approaches in graphics and multimedia [3]. In particular, the use of delegation, and the notion of "trait" objects used for example in the SELF system [83] are attractive from the viewpoint of building highly adaptable and extensible systems. However, technical issues aside, the fact that prototype models are strongly bound to experimental systems, and are not in widespread use, represented a serious barrier to their use within PREMO. The result is that the PREMO object model is based from the outset on a fundamental distinction between objects and classes, which in PREMO are called "object types". A detailed account of this model is given in Chapter 3; following an informal overview of the model, the remainder of this section describes other high-level design decisions that affected the content of this component.

### 2.3.1 Overview

A PREMO system consists of a collection of objects, each with a local (internal) state, and an interface consisting of a set of operations. Each object is an instance of an object type, which defines the structure of its instances. An object type can be defined as an extension to one or more other object types through inheritance; note that this allows for multiple inheritance. An important property of the model is that objects are never accessed directly. Instead, a PREMO client requests a facility called an "object factory" to generate an object satisfying specific criteria, and if it is able to comply, the factory will return a handle to the new object called an object reference. All subsequent activities involving the object is then done via the reference, for example invoking an operation on the object, or passing the object as a parameter to another operation. This separation of objects (i.e. physical storage) from their references is needed to support the aim of distribution, as an object reference can be used to encode both local address information and the location of a particular object across a network.

### 2.3.2 From Language Bindings to Environment Bindings

Although the choice of a class, rather than object-based model is relatively straight forward, a number of further options are rather less apparent. In particular, the aim of making the standard language independent introduces a tension in the design, between introducing features that offer descriptive or computational power but are specific to a restricted set of languages, or using a simple, less powerful model to describe the standard in the expectation that it will be easier to map the model onto the facilities of a given implementation language. Features that are problematic range from the mundane, for example how (or even whether) objects are copied, to complex problems such as the management of remote or distributed objects.

One approach that PREMO employs to prevent over-commitment to a particular object model is to introduce the notion of an environment binding. Previous standards in computer graphics have also been developed using a language independent description, and have been mapped onto a specific implementation language through a language binding, that associates the abstract data types and operations defined in the text of the standard with concrete data types and operation signatures within the target language. Such a binding is still needed for PREMO. However, while some concepts in the standard will be mapped onto language-specific features (for example, object types and operations), other aspects of the model, for example how objects are to be copied, or how remote objects are accessed, are left as facilities to be provided by the *environment* of a PREMO implementation. These facilities may be realised through language constructs, but more generally they may be provided by library packages, or even via the use of other standards. Thus, access to distributed objects within a C++ implementation of PREMO could be realised through a custom-built mechanism, or through a separate standard such as CORBA [71]. In the case of a Java implementation, these two options again exist, but in addition it is possible to use the Java RMI package [39]. By viewing features such as object copy and remote access as requirements on the environment, rather than requirements on the object model, the object model itself is simplified and is consequently easier to map against the provisions of a specific implementation model.

### 2.3.3 Object References

It is widely accepted that a fundamental component of object orientation is that each object in a system has an identity that is independent of that object's state. Therefore, two objects that have the same state can nether the less be distinguished. At a very practical level, this corresponds to the use of pointers to reference objects within an implementation. These pointers, or object references, may be implicit or explicit. In the case of SmallTalk or Java, for example, it is not possible to access an object other than through an object reference - this is enforced in the definition of the languages, which provide no constructs for referring to an object other than through pointers. C++ and Ada'95 have a different model. Objects in these languages are defined as generalised records, and a pointer to an object is a well defined type that is quite distinct from the type of the object itself.

As PREMO objects can be distributed, various mechanisms for accessing objects may be used within even a single system. For example, local objects might be referenced via pointers, while remote objects are referenced by some form of extended URL. To avoid confusion and implementation bias, the standard introduces the concept of an object reference as an explicit part of the object model, with the intention that this be bound to whatever means are used within the target language or environment to access specific objects. The approach taken in PREMO combines elements of the explicit and implicit approach. In line with the former, the model defines both the concept of an object, and an object reference. However, the distinction is there to simplify the use of multiple implementation strategies — it is not possible to refer to, or use, an object directly. Instead all access to an object, for example to invoke an operation, must be via an object reference.

### 2.3.4 Active Objects

Concurrency is by definition an integral aspect of multimedia presentation, and will certainly be a property of the type of distributed application which PREMO is intended to support. Fundamental to such a model is the idea that several threads of control, or processes, can be active within a system at one time, and that such processes interact through communication events. Here again there is a tension between adopting a simple model based on a particular set of facilities, or a more general model that is harder to use within the standard but is hopefully easier to implement.

On the one hand, there is a natural and appealing parallel between the idea of a process and that of an object. A process is an entity which encapsulates a thread of control and that interacts with its environment through events; an object is an entity that encapsulates state and interacts with its environment through operations. Languages such as Eiffel [66] and more recently Java have built on this view by treating processes (or threads) as particular types of object; in Java for example, an object will be active if it implements the *Runnable* interface. In contrast, other languages have maintained a separation between these concepts. In Ada'95 for example, processes are realised through a sophisticated task model, quite separate from the notion of task, while in C++ there is as yet, unfortunately, no standard model for dealing with processes.

The PREMO object model assumes that all objects are conceptually active; as we will discuss in section 2.4.1, the standard does however, for efficiency reasons, define certain types of objects to have trivial activity. What the standard does *not* do is to mandate any particular mechanism through which object activity should be realised. What is required is that each object has the capability to have an internal thread of activity. In parallel with this internal activity, an object may receive requests for an operation to be invoked; these requests arrive at operation receptors. At any time an object can select which requests it is willing to service. The PREMO object model does not place any requirements on the execution order for operations, for example pending requests may be serviced sequentially or concurrently.

### 2.3.5 Operation Dispatching

The delays inherent in remote object access and operation invocation mean that asynchronous operations are a fundamental tool in the development of distributed systems. Synchronous operation calls, in which the caller is suspended until the called operation terminates, are also required. To support multimedia applications, the design of PREMO also allows for a third kind of operation, sampled. A sampled operation is similar to an asynchronous one, because once the operation has been invoked the caller is able to continue its processing while the request is held in a queue. The difference is that the queue of requests for a sampled operation is effectively a one-place buffer, with any request for the operation overwriting any pending request.

Each PREMO operation is defined as using one of these *operation request modes*. The existence of these modes is one of the more significant differences between the PREMO object model, and that found in most programming languages, or indeed the model defined by the OMG.

### 2.3.6 Attributes

One of the positive aspects of object orientation is the emphasis on data hiding and encapsulation — clients of an object should only use the operations in the interface of an object, and should not have access to the internal state. Instead, if access to a variable is required, it should be realised through operations that retrieve or set the value of the variable. A number of such state variables appear within PREMO object types, and rather than define explicit operations for manipulating these variables, the standard introduces the concept of an *attribute*. The definition of an attribute looks like that of a variable, however an attribute of an object type is understood as being a shorthand for a pair of operations in the interface of that object type which set and get the value of an (internal) state variable. An attribute can be declared as read–only, or write–only, meaning that the corresponding 'set' or 'get' operation is not available.

### 2.3.7 Non-object Data Types

SmallTalk was for some time presented as the prototypical object-oriented programming system, and many of the ideas it pioneered were adopted in subsequent languages and systems. One of its strengths was its simple ontology; everything in the system is presented as an object, even "atomic" data such as numbers and characters. While this view produces a remarkably uniform model, it does have a number of consequences. First, there are a number of general raised by such an approach, including how one interprets the "identity" of numbers, how one relates binary operations on data "objects" to the conventional mathematical view of numbers. Second, there is the issue of efficiency: treating data values as objects implies that operations such as addition are handled by the same run-time dispatch mechanism as other operation calls. Data processing in computer graphics and multimedia often involves a considerable amount of numerical processing with large data sets (geometric structures, digital image formats, etc.) and here the need to use a general dispatch model is clearly an efficiency concern. Finally, while PREMO is intended to be language independent, the most likely targets for a language binding were seen as the family of object-oriented languages, including C++, Ada'95 and Java, in which object-oriented structures have been added to a language in which primitive data are treated as values. For these reasons, PREMO has adopted a model that distinguishes between non-object data types, such as integers and characters, and object types.

## 2.4 The Foundation Component

The implementation of most multimedia systems involves a number of fundamental concerns: control and management of progression through media content, synchronisation between activities, time, and coordination. Existing standards provide specific facilities for some of these tasks, while for others an implementor may need to utilise a general library (for example, for synchronisation) or develop ad-hoc solutions. Without mandating any specific approach to these general concerns, the PREMO Foundation component provides a set of general-purpose object and data types that can be used by a developer to implement the functionality mentioned above. A developer can either use these facilities "raw", to create a customised architecture, or they can be used via the higher level object types and services provided by Parts 3 and 4 of PREMO which are described later in this chapter. Because the Foundation component is essentially a toolkit, the remainder of this section describes its main provisions in terms of the principle media system requirements that are supported.

### 2.4.1 Structures, Services, and Types

The requirement that, conceptually, all PREMO objects are active means that in principle all access to an object must allow for the possibility that the object will have its own thread of control. Depending on the implementation platform, this assumption may impose a high overhead on the cost of accessing components of objects; such access will for example have to pass through the operation receptor and request handling infrastructure. For some aspects of media processing, these overheads are unavoidable; they are needed to support the provision of distributed services across a media network. However, in a typical media application, not all objects will necessarily be used as "active" entities that provide services. One use of objects is as data encapsulators, similar to the use of records (structures) in languages such as Ada and C. There is clearly a trade-off here, between the elegance and simplicity of a homogeneous object model on the one hand, and the practical problems involved in storing and processing large multimedia datasets on the other. For example, a visualisation application may need to operate on a volume data set containing in the order of $10^7$ - $10^9$ voxels [74]. If each voxel is represented as an object, the overhead in processing this dataset will become significant.

PREMO has adopted an approach that retains a fundamentally simple object model while allowing implementors to avoid the overhead of the full operation request system where it is not required. The approach is based on the top-level organisation of the PREMO object type hierarchy shown in Figure 2-1. All object types in PREMO are subtypes of `PREMOObject`, in which fundamental object behaviour, such as the ability of each ob-

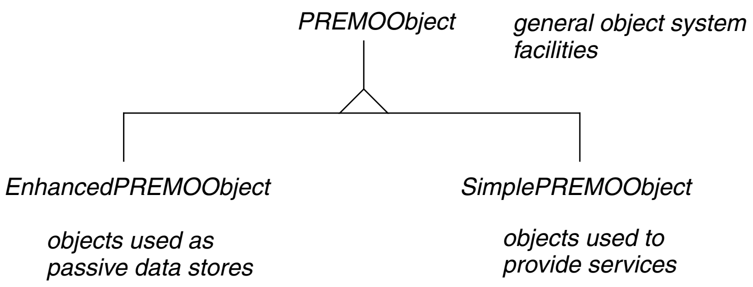

PREMOObject
general object system facilities
EnhancedPREMOObject
objects used as passive data stores
SimplePREMOObject
objects used to provide services

Figure 2-1 — Two Kinds of Object Type in the PREMO Hierarchy

ject to return information about its type, is defined. Below this the hierarchy bifurcates. `SimplePREMOObject` serves as a supertype for those object types that represent data encapsulators. Such object types are referred to as *structures*. `EnhancedPREMOObject` is the abstract supertype for those object types that provide services, and which therefore incur the overhead of the operation dispatch mechanism. This separation is further formalised through the profiles that are defined in each component to identify those object and non-object types that should be made available to clients of the component. Each profile consists of lists of object types, either under the category “provides type”, or “provides service”. Only an object type that inherits from `EnhancedPREMOObject` is allowed to appear in the “provides service” clause, and it is only objects of these types that a client can expect to interact with through operation dispatching.

### 2.4.2 Inter-Object Communication

Although ultimately all interaction between objects within a PREMO system takes place via operation requests, this is not a particularly useful way of representing communication and cooperation within a distributed system. In the case of multimedia, two models are now well known:

- Stream-based models, in which information related to processing is sent on communication channels or media streams between objects. These may be the same streams that are used to carry media data.
- Event-based models, in which there is conceptually a separate mechanism by which specific operations in the interface of a collection of objects can be invoked in response to a specific situation in one object.

PREMO does define media streams that in principle can be used to support communication between objects. These are described in section 2.5.3. However, streams are a comparatively “heavyweight” facility, intended primarily to manage the transport of media data. Instead, the foundation component defines a collection of object types that provide an event management facility for inter-object communication.

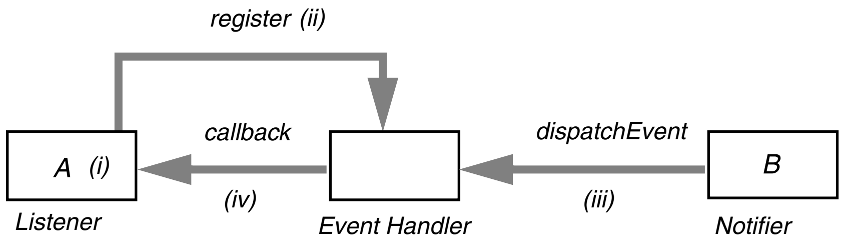


Figure 2-2 — Overview of Event Management

The event mechanism is based on callbacks and event handlers. Callbacks are now widely used in the graphics and user interface management communities, having been popularised through systems such as the X library [77], OpenGL, and more recently the Java AWT [28]. Essentially, a callback is just an operation in the interface of an object that will be invoked by some other entity within a system in response to an event. A typical low level example is an operation in a user interface object that a run time system will invoke to notify the object of a mouse-button being pressed or released. Callbacks often take parameters that carry information about the event that has taken place. Since the event management facilities in PREMO are used to address a range of concerns, it was sensible to introduce a systematic approach for carrying event information. To this end, an `Event` object types is defined to carry such information, specifically the name of the event, a reference to the source of the event, and additional data specific to the event.

Figure 2-2 provides an overview of the approach. Objects that are interested in a particular event, (object A in the figure) must (i) be of a type that inherits from the `Callback` object type, which provides a general `callback` operation, and (ii) must register their interest with an instance of the `EventHandler` object type. When an object (B in the figure) wants to notify the system that an event has occurred, it invokes the dispatchEvent operation on an event handler (iii), and all objects that have registered with that handler to be notified of the event will have their callback operation invoked (iv).

In fact, chains of event handlers can be set, because the `EventHandler` object type itself inherits from `Callback` and defines its `callback` operation to have the same effect as `dispatchEvent`. Thus, object A in the figure could be an event handler that subsequently distributes the event received by the callback to further objects.

In the case of a basic event handler, objects are only required to register with the handler if they should to be notified of a particular event; any object in the system can signal to the handler that such an event has occurred. A specialised form of event handler, called an `ANDSynchronizationPoint`, provides a richer service. Objects not only register to be notified of an event, they also register as *notifiers* for a particular kind of event. When appropriate, a notifier signals the event handler as usual, however, the event handler postpones the notification of objects interested in the event until *all* objects that have registered as notifiers have signalled the event to the handler. This object type has a role in the general synchronization facilities of PREMO, which are discussed next.

### 2.4.3 Synchronization

Like event handling, synchronization requirements in PREMO span a range of levels. At the level of data streams, fine-grained synchronization may be used to implement quality of service requirements, for example maintaining an adequate alignment between related audio and visual content. At a higher level, a multimedia presentation will typically consist of a collection of components, some of which may be presented in parallel. In addition to any fine level of synchronization between such strands, synchronization between key milestones (such as the start/end of component strands) may be required. Beyond direct control of media presentation, synchronization may also be needed within the general control structure that manages the overall media system.

Synchronization in PREMO is supported at two levels - in terms of *events*, and in terms of *time*. Event-based synchronization has obvious application in dealing with the processing of structured presentations composed of more primitive media streams, however it also has a role in synchronizing the presentation of the data within a stream, where significant milestones are defined by the content of the stream, rather than its absolute position. An example of this is the synchronization of ultrasound or other medical scan data, where milestones defined by physiological events need to be aligned. Such an example is described in more detail in [63]. Time-based synchronization is better known, and involves ensuring that multiple activities reach particular milestones at times specified relative to each activity.

The event and time-based approaches are both supported by a common framework, the *Synchronizable* object type, which PREMO uses as the basis for representing, monitoring and controlling the transmission and processing of media data. Although the interface to this object type is large, it is based around three main ideas:

1. An internal progression space, which acts as a coordinate system for defining the concept of location within some media stream or content. Synchronizable objects do not themselves carry media data, but instead are inherited by object types which are involved in the transport and processing of such data. Conceptually, the progression space represents the temporal extent of some media representation, and progress through the progression space is made during processing of that media.

2. Progression is controlled by a finite state machine. This is actually achieved by having `Synchronizable` inherit from another object type, called a `Controller`, which is also defined in this component. Controllers are described in Chapter 5 and their details are not of concern in this overview. It suffices at present to say that a Synchronizable object can be in one of four states: stopped, playing, paused, and waiting. Conceptually, when an object is in the playing state, progress is being made through its progression space. Transitions between the states occur as a result of operation invocation, and also through interaction with reference points, which are discussed below. A number of attributes define the parameters that affect how progress is made, for example, the direction of progression.

3. *Reference points* can be placed along the progression space, either individually, or repeated with a given period. Each reference point consists of an event, a reference to an event handler, and a special boolean ‘wait’ flag. When a reference point is

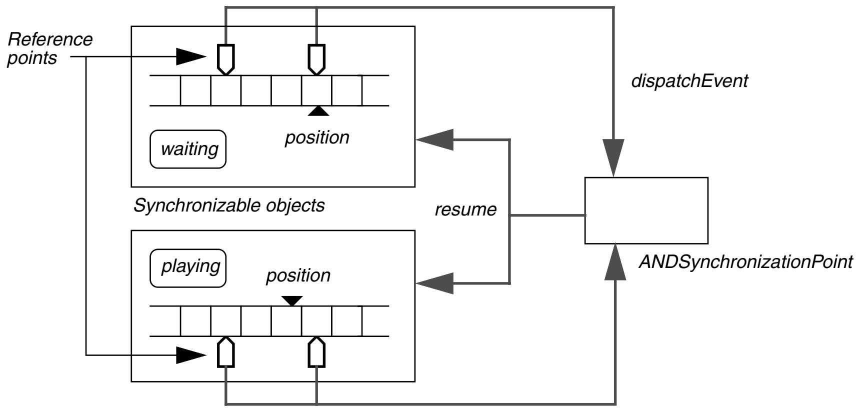


Figure 2-3 — Example of a Synchronization Scheme

encountered during progression, the event is sent to event handler specified. The wait flag indicates whether progression should be suspended at this point, and if has the value *true*, the *Synchronizable* object is placed into the 'waiting' state, where it will remain until the *resume* operation in its interface is invoked.

Reference points and the 'wait' flag are intended to be used in conjunction with other PREMO facilities to implement synchronization schemes. For example, by combining reference points with the `ANDSynchronization` object type described in section 2.4.2, processing of one part of a presentation can be suspended once a particular milestone has been reached until all other `Synchronizable` objects that involved in implementing the presentation have reached related milestones. An example of such a scheme is shown in Figure 2-3.

### 2.4.4 Time

Media such as sound, video and animation is fundamentally grounded in time, and to describe and control the presentation of such media it is necessary to have some means of representing and measuring time. The question of how time should be represented (for example, as a continuum, or discretized) has been the subject of much philosophical debate, and is a non-trivial concern in areas such as real-time systems modelling and verification. PREMO adopts a pragmatic approach, in which all representations of time are based on 'ticks' produced by some clock. The granularity of a 'tick' is not fixed by the standard, but rather depends on the particular clock used.

PREMO introduces object types to represent abstract clocks, a subtype of clocks representing 'real time' system clocks, and a resetable timer. All clocks are derived from the abstract object type `Clock`, and specify a 'tick unit', which is the unit (for example, seconds) represented by each tick, and a measure of the accuracy of the clock. An actual measure of time is obtained by invoking the `inquireTick` operation in the interface - however, it is up to subtypes of `Clock` to attach a meaning to the number of ticks that are returned. Thus an object of type `SysClock` returns the number of ticks (to its level of accuracy) since the start of the defined PREMO era. The object type `Timer` defines a start/stop timer by extending the interface of `Clock` with operations for stopping, starting, and pausing the progression of time. For objects of this type, the number of ticks returned by inquireTick are the number of ticks that have elapsed, while the object has been in its running state, since it was started (i.e. ignoring time spent in the pause state).

The link between time, and the event-based synchronization model described in section 2.4.3, is defined by the object type `TimeSynchronizable`, which couples the behaviour of a `Synchronizable` object with that of a `Timer` object, thus making it possible to measure and control the `speed` of progression through the internal span of a synchronizable object. The interface of `TimeSynchronizable` allows reference points to be placed against positions on the progression space specified in terms of time, for example, placing a reference point 30 seconds from the start of a video sequence. Obviously, the actual point in the video content at which this reference point will be reached will depend on the speed at which progression is being made through the video. Two subtypes of `TimeSynchronizable` are identified in the standard. A `TimeSlave` object is one for which the rate of progression can be 'slaved' to the rate of progression of some other time-synchronizable object. A `TimeLine` object can be used to set reference points against milestones in real time.

### 2.4.5 Property Management

In the PREMO object model described in section 2.3, the attributes and operations of a type are defined statically, when the object type itself is defined. Once an instance of a type is created, the interface of the object is fixed. This "static" approach to object structure has clear benefits, not the least being support for compile-time checking that can reduce the likelihood of programmer error. However, as we mentioned in the introduction to section 2.3, more dynamic object models are also available, and their potential use in graphics and multimedia has been noted [3,33]. Features such as delegation, or on a more modest level, the ability to alter the interface of an object at run time (as adopted in Python[87] for example) would play a useful role in the implementation of constraint management for example. However, the experience of the MADE project [42] was that implementing such features within a class-based, 'static' object models was a significant problem.

PREMO introduces the concept of object *properties* as a compromise between a purely static model and the facilities offered by dynamic models. A property is a pair, consisting of a key (i.e. a string) and a sequence of values. Each value in the sequence can come from any PREMO non-object data type, and as these include object references, an object property is essentially a dynamically typed variable. The `EnhancedPREMOObject` type introduces operations to define, delete, and inquire values associated with a given property key. Later in the book we will see how properties can be used to implement various naming mechanisms, store information on the location of the object in a network, create annotations on object instances, and underpin a framework for inter-object negotiation. In support of this, the standard stipulates that objects of certain types will have a property with a given key, and possibly particular values. However clients of any object whose type inherits from `EnhancedPREMOObject` can attach new properties at any time. Properties may also be declared as 'retrieve only'.

The basic facilities provided by `EnhancedPREMOObject` are developed by two further object types, `PropertyInquiry` and `PropertyConstraint`. In the first of these types, each property key can be associated with a corresponding 'native property value', which describes the range of values (capabilities) that the corresponding property can take on. This can be viewed as a form of dynamic typing. The `PropertyConstraint` type extends this model by ensuring that a value added to a property lies in the corresponding native property value, if this exists. This object type also introduces a number of 'meta' properties, for example, the key '`dynamicPropertyListK`' is associated with a list of values representing the keys of certain properties. The operations bind and unbind allow keys to be added to and removed from the values of `dynamicPropertyListK`. Only while a property's key appears under this property can the corresponding value be changed.

### 2.4.6 Object *Factories*

One specific use of properties is in the creation of objects. In section 2.3.2 we noted that PREMO relies on its environment to provide certain fundamental services, and the creation of objects is one such service. In most object-oriented programming languages, creation is a comparatively simple mechanism, handled either by a language construct (e.g. the '`new`' operator of Java) or through some meta-object system, in which classes are themselves objects and can respond to message requesting object creation, as in SmallTalk. This situation is complicated in PREMO by the use of properties to describe features of objects. For example, a PREMO system may define a JPEG decoder as an object type that has a property, say "`GIFVersionK`" which can be set to either the value '87a' or 'v89a' (CHECK) representing the two standard specifications. Alternatively a system may offer two types of GIF decoder object, one for each version of the format, in which the property "`GIFVersionK`" is fixed. There is thus interaction between the structure of the type hierarchy, and the use of property keys.

In fact, from the viewpoint of a PREMO client, the specific type of an object will often be uninteresting. What is important is (i) that the object is a member of a subtype of a given type, and (ii), that the properties of an object satisfy a given constraint. In the example above, what the client may really want is a device that can decode JPEG v87, and the client is not concerned whether this device is an instance of an object type specifically for this version, or is an instance of a more general object type that can be configured to the given requirement.

In order to hide these issues, and provide a uniform interface for object creation, the foundation component of PREMO introduces the concept of an *object factory.* A factory is itself an instance of the `GenericFactory` object type that provides a single operation, `createObject`. This operation accepts an object type, and a set of constraints in the form of a sequence of key / permitted value pairs, and (if possible) returns a reference tp an object that is an instance of the given type or a subtype, and whose properties satisfy the constraint.

Factories are themselves objects, and a PREMO system provides a factory finder object that is able to locate a factory capable of producing an object that will meet given constraints.

## 2.5 The Multimedia Systems Services Component

Multimedia systems typically integrate a variety of logical and physical devices. For example input and output might involve devices such as video cameras, microphones, and a sophisticated speaker system. Processing in turn may involve logical devices such as data encoders/decoders, media synthesizers (e.g. a graphics renderer), and a video mixer. The data produced an consumed by these devices takes a variety of forms, for example a discretised audio signal, a sequence of video frames, or a discrete graphics model. In turn, these forms can be encoded in a variety of formats (ALAW and ULAW for audio, for example). Finally, different protocols may be available to communicate such data, depending on the source and destination hardware, and on the available network infrastructure.

As explained in the introductory chapter, PREMO does not aim to define new standards for the encoding or transport of media data. Rather, it seeks to provide a set of facilities that abstract away from the details of low level system services, instead providing an application developer with a uniform high level view of media processing. To this end, the multimedia systems services (MSS) component of PREMO defines the infrastructure for creating and maintaining a network of heterogeneous processing elements for media data. This includes object types for describing generic resources, devices, and facilities for organising a collection of such components into higher level units with a single interface. MSS encompasses mechanisms by which media processors can advertise their properties for network construction, can be interconnected and controlled, and can be configured dynamically to match the needs of a network while in operation.

MSS was originally defined by the Interactive Multimedia Association [45], a large consortium of industrial vendors and developers. IMA were aware of the work within SC24 on the development of PREMO, and donated the MSS framework to the Committee. It was subsequently adopted by SC24 as the basis of a distinct PREMO component. During the development of the standard, several of the main provisions of MSS were refined and integrated with facilities from the Foundation component.

### 2.5.1 The Paradigm of Media Networks

In order to abstract away from the details of specific media types, media processing elements are viewed as “black boxes” that can be interconnected through a high-level interface to construct a network of such elements appropriate for a given application. At this level, a PREMO application using MSS resembles a dataflow network, where the nodes correspond to media processors, and the data streams carry media content. The adoption of a dataflow-oriented view of media system architecture is not peculiar to

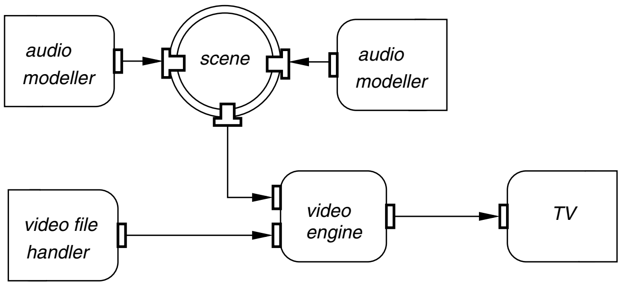


Figure 2-4 — Simple Multimedia Network

PREMO. It has appeared in published approaches to multimedia systems (for example, [34]), and is also increasingly used in "plug and play" applications environments, for example the visualization tool AVS [84] uses such a model for interactive construction of applications from a toolkit of basic modules.

Figure 2-4 contains an example of a small network. It represents a video engine combining input from a local file (for example, in MPEG) with audio clips stored as media primitives within a remote database (scene). The audio primitives in the scene are constructed by a number of audio modellers (MIDI devices, or waveform editors, for example). The combined audio/video output is presented on a TV device.

The devices in the figure are all instances or subtypes of specialised object types defined in the fourth component of PREMO, and which is discussed in section 2.6. What makes the construction and operation of such a network possible are that all of the object types involved extend the virtual device and resource concepts defined in Part 3 of PREMO. This allows the devices to be connected together, and subsequently to exchange media data along the streams shown. In the remainder of this section we describe the principle concepts and types that the MSS component provides for the creation of such networks.

### 2.5.2 Virtual Resources

A high level view of a media network is of a collection of resources that cooperate in the task of creating or processing media. These resources encompass physical devices (such as cameras or mixing suites), software processes such as graphics renderers and audio filters, as well as supporting infrastructure such as connections and software for managing collections of lower-level resources. What is fundamental to this view is, first, that a resource is something that has to be acquired for a task, and second, that many of what we consider to be resources are inherently configurable. For example, an audio mixer may involve both hardware and software elements, access to which must be acquired before the mixer can be installed in a processing network. In fact, a number of mixers might potentially be available, differing in characteristics such as the number of channels that they can accept, the kind of audio formats that can be processed, and the type of filters that can be applied.

The property description and management facilities described in section 2.4.5 form the basis for realising this model. The characteristics of a particular resource are described by properties; some of these can be set by a client of the resource, often to one of a set of possible values defined as the native property values for the given key. Other properties, representing immutable aspects of a particular resource (for example the number of input channels to the audio mixer) are read only, but still play an important role in establishing a media network.

The fundamental operation of a PREMO resource is defined by the `VirtualResource` object type. Each resource (or more generally, each subtype of `VirtualResource`) defines a set of property keys and values that are relevant to the description and control of the resource. In addition, each resource encapsulates a number of *configuration objects*. These objects store data about the resource to which they are associated, and this information is used by other objects, for example in providing communication services or quality of service management. The MSS component defines three types of configuration object explicitly; each inherits from `PropertyConstraint`:

- `Format` objects represent the details of a media format, for example the organisation of a bitstream;
- `MultimediaStreamProtocol` objects provides information about how media data is conveyed between processing nodes; and
- `QoSDescriptor` objects capture quality of service characteristics, such as the level of guaranteed service, and bounds on delay and jitter.

It must be emphasised that the PREMO standard does not describe all details of these object types; for example the specifics of particular media stream formats. The purpose of these object types is to provide placeholders and hooks that can be specialised or used as required within a particular implementation environment. What the `VirtualResource` object type does provide are operations for accessing particular configuration objects using semantic names (strings), acquiring the physical and software resources managed by the object, and validating whether the configuration requirements expressed by the combination of properties and configuration objects can be satisfied. Each resource is also associated with a stream control object, the purpose of which is described next.

### 2.5.3 Stream Control

Virtual resources are involved in the production and transport of media data. Control and monitoring of media streams is provided in PREMO by the `StreamControl` object type. Different kinds of resource will have different views on media streams, ranging from a low-level signal oriented view, through levels that abstract signals into packets, and packets into media samples or chunks. This range of views is accommodated by basing stream control on the `TimeSynchronizable` object type discussed in section 2.4.3; by inheriting from this type, stream control can be defined with respect to the coordinate system of the progression space, or (relative) time. To facilitate fine control over progress, the `StreamControl` object type refines the state machine inherited from *Synchronizable* by introducing states that allow media content to be drained (dis-

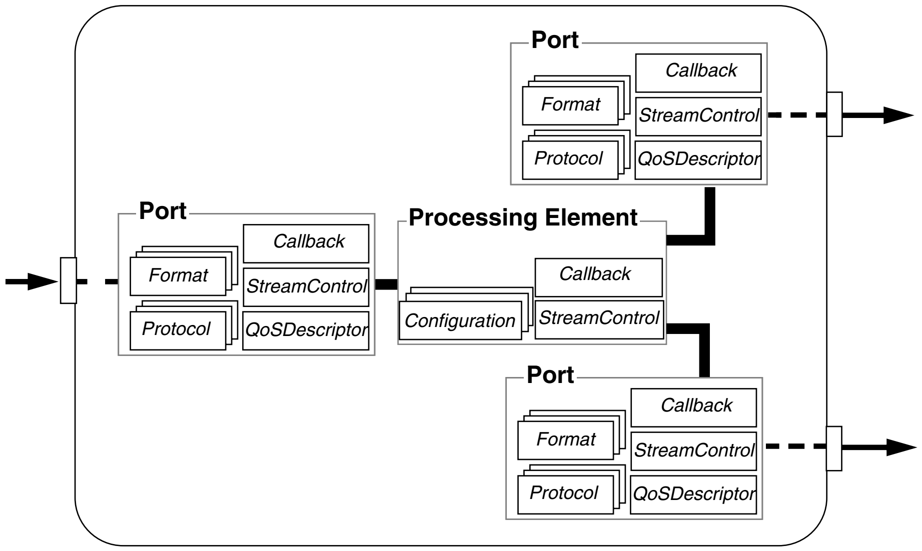


Figure 2-5 — The Structure of a *VirtualDevice* Object

carded) or buffered and subsequently released. These facilities, along with the ability to place reference points along the progression space connected to the event handling system, are intended, for example, for use as part of an overall quality of service management strategy. A further object type, `SyncStreamControl`, allows progression through its stream to be synchronized (slaved) explicitly with the progression of some other object that is derived from the `Synchronizable` type.

Virtual resource objects have an associated `StreamControl` object that allows, where applicable, monitoring and control of the end-to-end processing carried out by that resource. Stream control objects are also a feature of an important kind of resource, the virtual device.

### 2.5.4 Virtual Devices

The "nodes" in the dataflow network shown in Figure 2-4 are defined to be so called *VirtualDevice* objects that form the basic building block for interaction and processing capabilities within PREMO. The anatomy of a virtual device is shown in Figure 2-5.

The principle features that the VirtualDevice object type adds to a resource is the presence of "openings", called ports, which act as input or output gateways for the virtual device, and the concept of a "processing element". Ports are the means by which data can be passed from one device to another. A port is not itself an object, rather, it an identifier or handle that is used to reference a particular opening, and through the interface of a virtual device, access and control information about that opening. Like a resource (and hence a device itself), each port is associated with a collection of configuration objects that characterise the flow of data through the port. More specifically, each port has associated `QoSDescriptor`, `Format`, and `MultimediaStreamProtocol` objects. The client can set the properties of these objects, and can refer to them when configuring a network. These configuration objects are combined into a port configuration object, which also contains a reference to an event handler dedicated to that port, and a `SyncStreamControl` object that controls and monitors the transfer of media data via the port. Just as with `VirtualResource`, an operation is provided by `VirtualDevice` to validate the requirements captured by the configuration of each port.

The "processing element", shown in Figure 2-5, is a conceptual, rather than concrete, component of a virtual device. That is, there is no object type for a processing element, nor does the `VirtualDevice` introduce variables or operations to implement it. The only part of a virtual device that directly relates to processing is the end-to-end stream control and configuration objects inherited from `VirtualResource`. One of the tasks to be addressed in implementing the `VirtualDevice` type is to decide how the transfer of media data within the device is to be effected. By *not* being prescriptive about this aspect, the PREMO designers have sought to better accommodate the wide range of existing media processing software that might be "wrapped" within a subtype of `VirtualDevice` for use in a PREMO–based network. Chapter 6 demonstrates one approach through which this interaction can be realised.

### 2.5.5 Virtual Connections

The lines in Figure 2-5 entering and leaving device ports represent the flow of media along streams. PREMO itself does not define a "Stream" object type, since much of the detail here depends both on the underlying network technology, and the context of the connection (i.e. whether two devices are on the same host, local network, etc.). Streams however are established and maintained by objects derived from another subtype of `VirtualResource`, the `VirtualConnection` type. As a resource, a virtual connection object contains a stream control object that represents the end-to-end flow of media data along the stream controlled by the connection. A subtype of VirtualConnection supports multicasting, with operations to attach and detach a device/port combination to and from the connection. All connections are unidirectional.

If the underlying devices are located on the same hardware, a connection may be realised by directly linking the input and output ports of the associated devices. More generally, the devices will be on distinct, possibly remote, machines and using different local facilities for inter–object communication. In such cases a virtual connection may need to create a *virtual connection adapter*, that provides appropriate interfaces to the end-parties while managing any recoding or translation of raw data required. Connection adapters exist only as concepts within the PREMO standard. They do not correspond to any particular object type, and in fact their implementation will in general require a collection of objects to manage the transfer between the different protocols.

### 2.5.6 Higher-Levels of Organization: Groups and Logical Devices

Even the simplest non–trivial media network, involving two devices with a single connection between them, involves a significant number of objects: the devices themselves, the connection, the connection adapter (if needed), event handlers for the ports and devices, and possibly supporting objects to, for example, monitor quality of service. For a realistic application, the number of objects is significantly greater, and the problem of tracking which particular groups of objects are relevant to any given part of the network becomes significant.

To prevent organizational anarchy, it is often convenient for clients to interact with a single object that represents each "significant component" of a network. PREMO provides a `Group` object type to support management of a collection of devices and connections. *Groups* are resource objects which control a number of other virtual resources (in particular devices and connections), and their respective network. By default, the constituent devices remain hidden to the external client; instead, groups provide a single entry point for stream control, as well as other services. By using the basic group interface, the client does not have to know about the interfaces of these constituent devices. Because Group inherits from `VirtualResource`, each group is itself a resource, and consequently, the configuration of group components can be validated, and the components themselves acquired, via the one group interface, rather than individually. Because a group is itself a resource, a group can itself be a member of a further group.

Although groups can be organized into hierarchies, it is important to remember that a group is not a device; it has no ports of its own. Instead, a client using a number of groups is responsible for ensuring that, where necessary, components of distinct groups are connected. A specialised form of group, called a *logical device*, combines the central resource management capabilities of a group with the processing model of a virtual device. Resources are added to and managed by a logical device in the same was as a group, but the client of a logical device can also dynamically define ports on the interface of the device. When defined, each port on the logical device is associated explicitly with a port on a device that it manages. A logical device thus acquires input and output ports, and can be built into a network in the same way as other devices.

### 2.5.7 Working in Unison

At this point we have described the main features of the multimedia systems services component. Given the importance of this component to the aim underlying PREMO, of supporting the development of distributed multimedia systems, it is useful to summarise the roles played by the various object types described here. We do so by outlining the steps involved in setting up a network using MSS.

1. Assuming that the client has a suitable factory, it first uses the factory to create the various objects (devices, connections, and other resources) that make up the network. Part of the specification for the objects given to the factory may involve constraints on properties of the objects, for example a device is able to receive data using a particular format.

2. The connections are defined by sending each connection object a *Connect* request, specifying the source and destination device/port combinations.
3. The client may create a group object, and then add all of the resources to the group by sending the request `addResourceGraph` to the group. At this point the structure of the network has been fixed, but no actual resources (e.g. bandwidth) have been allocated to it.
4. Using the acquireResource request of the group object, the client attempts to allocate the resources needed for each of the objects in the network. Inability of the underlying system to meet this request will result in an exception which the client can detect. In this situation it may modify its requirements by changing properties of any of the objects within the group, for example by settling for a less reliable connection.
5. Once the resources have been allocated, the client can start the transport of media data through the network by accessing the `StreamControl` object of the group.

## 2.6 The Modelling, Rendering, and Interaction Component

A feature of the MSS component is that its provisions are independent of the data processed by the devices and resources within a network. Thus the same approach can be used for setting up a video editing system as for setting up a virtual reality modelling and rendering environment. The fourth component of PREMO describes general facilities for the modelling and presentation of, and interaction with, multidimensional data that utilises multiple media in an integrated way. That is, the data may be composed of entities that can be rendered using graphics, sound, video or other media, and which may be interrelated through both spatial coordinates and time.

The MRI component is interesting for two reasons. It is the point within the PREMO standard where the actual structure and content of media data becomes significant. It is also the point at which ‘traditional’ computer graphics, i.e. modelling and rendering of synthetic scenes, is integrated into the broader concerns of multimedia. This integration within an object-oriented framework highlights a significant design issue regarding the implementation of graphics (and for that matter, other media) processing, which we discuss in the first of four sub-sections.

The actual description of the MRI component ranges over three concerns, which are each covered by a separate heading thereafter. Section 2.6.2 concerns is the design of a hierarchy of modelling primitives for characterising multimedia presentation. Section 2.6.3 deals with the collection of devices that extend the `VirtualDevice` type of the MSS component to allow modelling, rendering and interaction to take place within a media network of the kind described in section 2.5.1. Section 2.6.4 focuses on a particular device, the `Coordinator`, that plays a key role in mapping presentation requirements of media streams against the devices that are available for processing media.

### 2.6.1 Object-Oriented Rendering

A fundamental question that must be addressed within any object-oriented graphics or multimedia system concerns the allocation of fundamental behaviour, such as transformations and rendering, to object types defining media content within an API. Two quite distinct approaches emerge. The first is to attach behaviour to the object types that are affected by that behaviour. For example, geometric objects and other kinds of presentable media data can be defined with a ‘render’ method, with the interpretation that such an object can be requested to produce a rendering of itself. Such an approach can be extended to collections of presentable objects, and fits well with the concept of an object as a container for data along with the operations that manipulate that data. The second approach is to define objects whose principle purpose is to act as information processors, and which receive the data that they operate on as parameters to operation requests or through some other communication mechanism. In this case, a ‘renderer’ object would receive presentable objects as input through some interface, and produce a rendering of those objects via some output mechanism. From the discussion in section 2.5.4 it may already be clear that PREMO has adopted for the second option. Separating operations (in the form of devices) from the data that they manipulate may appear to violate a central tenant of object–oriented design. However, it has two important benefits for PREMO.

1. First, a direct and desired consequence of a distributed model is that one model or data set may be rendered by several processes working in parallel at various locations. It is difficult to see how this can be realised efficiently in an architecture in which each media object renders or processes itself. Either such objects must be able to support multiple concurrent threads internally, or any object that is to be rendered must first be copied. In contrast, treating renderers as a form of processing device means that multiple renderers can be created (relatively) easily to operate on a given database of objects representing media data (see for example Figure 2-4). Such a database can either be shared by several renderers, or there may be several copies of the data. Strategies for managing the distribution, update, and access control of data within such a system are well known, and thus this approach is rather more practical and flexible than the alternative.

2. It supports an approach to application development based on interconnecting a number of processing devices — irrespective of whether those devices are operating on continuous media such as video, or a series of discrete data sets within a rendering pipeline. Once such a network has been defined, it can be used for a variety of data sets or models, and can be readily modified. In contrast, in an architecture where (for example) graphics objects render themselves, the control of processing and flow of data is encoded within specific operations, making it difficult to develop an application that can be modified or extended without wholesale reprogramming of those operations.

By opting for a model in which media data is essentially passive, while media processors are active objects that provide services, PREMO aims to provide a uniform, integrated treatment of both digital media and synthetic graphics.

### 2.6.2 Primitives

The potential domains of application for a system such as PREMO are diverse. When considering the design of a component for modelling and rendering, this raises the difficult problem of identifying an appropriate set of ‘media primitives’ — or indeed, whether to include any model of primitives at all. Two directions initially appear feasible when considering how primitives for modelling and rendering could be supported in a system like PREMO. First, it would be possible to take an existing set of primitives from an established system, for example the node set provided by Open Inventor [89], and adopt these to the needs of PREMO, possibly through some further extensions. The problem here is in finding a set of primitives suitable for the range of applications addressed by PREMO, and then deciding on what, if any, extensions to include. The second approach is to derive some minimal framework of elementary primitives from which those used in practice can be derived by composition.

Although an interesting research problem, both this and the first approach are biased towards a model in which PREMO devices for modelling and rendering would effectively be implementing a new standard for graphics primitives. It is simply unrealistic today, given the investment in graphics and media technologies, to expect industries to adopt a new standard for media data. Instead, the philosophy underlying PREMO is to view the standard as a framework for supporting the integration of different modelling and rendering technologies, with their own models of media data, within a heterogeneous distributed system. This has already been reflected in the discussion on virtual devices, where we noted that the virtual device specification does not mandate any specific strategy for implementing the processing element, thus allowing existing media processors to be accommodated.

In this context, the role of primitives is rather different from their role in a detailed standard such as PHIGS [44]. PREMO clearly cannot attempt to describe a closed set of primitives for modelling and rendering. Instead, it defines a general, extensible framework that provides a common basis for deriving primitive sets appropriate to specific applications or renderer technologies. Graphics modellers, for example, may use specific representations such as constructive solid geometry, NURBS surfaces, particle systems etc. Audio modellers may use primitives that represented captured waveforms, or raw MIDI data for synthesis. The aim of the primitive hierarchy defined in this part is to provide a minimal common vocabulary of structures that can be extended as needed, and which can be used within the property and negotiation mechanisms of PREMO as a basis for devices involved in modelling and rendering to identify their capabilities for use in a network. The seven categories of primitive defined in PREMO are:

1. *Captured* primitives. These allow the import and export of data encoded in some format defined externally to PREMO, for example MPEG [55].

2. *Form* primitives. Here the appearance of the primitive is constructed by some renderer or more general media engine. These include geometric primitives (polylines, curves etc.), as well as audio primitives for speech and music, etc.

3. *Wrapper* primitives allow an arbitrary PREMO value to be carried as a primitive, for example for use in returning the measure of an input device.

4. `Modifier` primitives alter the presentation of forms, for example visual primitives encompass shading, colour, texture and material properties that affect (for example) the appearance of geometric primitives.

5. `Reference` primitives enable the sharing and reuse of clusters of primitives via names that can be defined within structures.

6. Forms and modifiers are combined within `Structured` primitives. An `Aggregate` is a subtype of `Structured` which contains a set of primitives, where some members of the set may be interpreted in application dependent ways; it is thus up to an application subtyping from `Aggregate` to impose a specific interpretation on such combinations. Of particular importance, given that PREMO is concerned with multimedia presentation, is the `TimeComposite` primitive and its subtypes which allow a time-based presentation to be defined by composing simpler fragments. Subtypes of `TimeComposite` provide for sequential and parallel composition, as well as choice between alternative presentations as determined by the behaviour of a state machine. Additional control over timing is achieved via temporal modifiers, and subtypes of `TimeComposite` define events that can be used within the PREMO event handling system to monitor the progress of presentation.

7. `Tracer` primitives carry an event. This event can be detected at the port of a device configured to use `MRI_Format`, and will be dispatched to the event handler associated with the port. This facility is used for coarse-level synchronization.

### 2.6.3 Modelling and Rendering Devices

The MRI component derives a number of object types from the `VirtualDevice` type of the MSS component, as described in section 2.5.4. As in MSS, these do not represent concrete devices. They instead define the interface that a device must offer in able to provide certain kinds of service within a PREMO system, and in the case of Part 4, with primitives derived from the hierarchy described above. The device network shown in Figure 2-4 on page 23 incorporates a number of devices, the types of which would inherit from MRI object types. The MRI component defines a subtype of `VirtualDevice` for use as the base type for deriving devices for modelling, rendering and interaction. The so-called `MRI_Device` object type is required to support a format at its input and/or output ports that allows MRI primitives to be transmitted and received. Such a device is also required to define properties setting out which primitives it can accept, and some measure of the efficiency with which it can process primitives. In the standard, the following specialisations of `MRI_Device` are defined:

1. *Modellers* and *Renderers* guarantee to provide an output or (respectively) input port that accepts `MRI_Format` streams for carrying primitives. The devices also contain properties that characterise their ability to process primitives.

2. A `MediaEngine` is a device that can act both as a `Modeller` and a `Renderer`, i.e. a device that can transform one or more streams of primitives into new streams.

3. The `Scene` object type defines a database that can be used to store primitives produced and/or accessed by other devices within a network. It is assumed, for exam-

ple, that multiple devices may have concurrent read access to specific primitives, but the exact form of concurrency control is not specified. The interface of the device allows requests for access to be granted or denied depending on the policies adopted.

4. Two devices are introduced to support interaction. The `InputDevice` object type (a mouse would be a concrete example) supports interaction in either sampled, request or event mode through the stream and event handling facilities defined in other parts of PREMO, while the `Router` object type allows streams of data to be directed based on the state of an underlying state machine.

When accessing primitives stored in a scene, or coordinating the processing of multiple media streams, it is necessary to be able to determine when a particular stream has been fully processed (or received, in the case of database access). This task is supported by the `Tracer` primitive, which carries a reference to an `Event`. Whenever such a primitive is encountered at the port of a device that is a subtype of `MRI_Device`, the event carried by the tracer will be dispatched to an event handler associated with the port. In this way, other objects that need to be aware of the progress of media processing can register interest in such events and be updated of processing activity.

## 2.6.4 Coordination

By using the primitives derived from the hierarchy described in section 2.6.2, an essentially declarative description of a multimedia presentation can be defined. Typically however, at some point this presentation will need to be processed or presented, and during this activity the internal structure of the presentation, for example as a collection of media data to be presented in parallel, becomes important. If a media network contains a device that can process such structures directly, the problem is solved. However, it is also possible that the presentation of a structured media primitive will require the services of multiple devices, whose activities must then be coordinated to reflect both coarse synchronization constraints, as well as quality of service requirements, inherent in the declarative model.

The MRI component defines a subtype of `MRI_Device` called a `Coordinator`. Such a device encapsulates a number of other media devices (derived from `VirtualDevice`), each of which provides the coordinator with one input port. The coordinator itself has one input port, and as it receives primitives in `MRI_Format`, the coordinator is responsible for decomposing any structured presentation into components that can be processed by the devices that it encapsulates. In the example, the coordinator may receive presentations that involve synthetic graphics, video, and audio components. The audio component of the presentation is delegated to the logical device responsible for audio rendering, while the graphics and video are managed by the second logical device. The coordinator is also responsible for ensuring that its components maintain any synchronization constraints captured by the overall presentation. It may achieve this by monitoring the overall end-to-end progression of its encapsulated devices, and placing synchronization constraints on those progression spaces, or by using more specific mechanisms available within PREMO or a given implementation.

## 2.7 Closing Remarks

This chapter has presented an overview of the PREMO standard. In the process, we have set out some of the design constraints that have determined the shape of the standard, and have discussed some of the alternatives that were considered. The description of PREMO object types and their behaviour has been, by necessity, incomplete and informal. In the chapters that follow, each of the four components will be presented in detail, including the specific interfaces defined for the object types mentioned here. By giving an up–front view of the overall provisions of the standard, it is hoped that the reader will be better able to relate the detailed account of the object types, as they are given, back to the overall picture of what PREMO is intended to achieve.

# Chapter 3

# The Fundamentals of PREMO

## 3.1 Introduction

To date, standards for computer graphics APIs have inevitably implemented in a low-level procedural language such as FORTRAN [82] or C [58]. The principle abstractions that these languages support are function and procedure headers, and type or constant definitions. An API standard could be described as a collection of data types and abstract procedures, which would then be mapped through a language binding into function or procedure definitions in a specific host language. Critically, there were few, if any, assumptions in the standard itself about how functions or procedures behaved. The main difficulty, at least in the early language bindings, was deciding how to cope with differences in the expressive power of the programming languages, e.g. FORTRAN did not allow enumerated type definitions, so any use of 'conceptual' enumeration types in the standard needed to map onto a set of constants within a FORTRAN binding.

At one level, PREMO represents a straightforward evolutionary step; rather than binding the standard to the facilities of a *procedural* language, the standard is intended to utilise the facilities provided by *object oriented* programming systems. A language binding for PREMO is able to map entities in the standard onto mechanisms such as classes, methods, and inheritance provided by an object-oriented language. However, this view misses a significant difference between PREMO and previous graphics standards. In the case of PREMO, the implementation technology, i.e. objects, and the mechanisms for their definition and interaction, are an intrinsic part of the standard. Before we can give a precise definition of the types and services of PREMO, it is first necessary to set out what we mean by terms such as 'object', 'object type', and 'inheritance'. Not only will these affect how we go about binding PREMO to a programming language, it will also affect how we construct complex entities in the PREMO specification from simpler ones.

The collection of definitions that set out the concepts with which PREMO is described form the first component of the standard, referred to as the Fundamentals of PREMO. This chapter describes the fundamental component, following closely the structure of the Part 1 document, but also explaining the rationale behind decisions, in particular those that have consequences on subsequent components and the reference implementation described in this book. Readers with a background in object oriented systems or modelling may find that they are familiar with some of the material and can skip sections. However, for at least issues (object references, and operation request modes) the approach taken in PREMO may be unfamiliar to larger sections of the audience.

For readers who have learned object-oriented programming or design from a particular language or notation, the definition of some of the concepts in this chapter may seem convoluted. It must be remembered that the PREMO object model is attempting to define a set of concepts that can be mapped onto a number of different programming models. Achieving this level of generality does unfortunately lead to some awkwardness.

## 3.2 Basic Concepts

All high-level programming involves the use of some paradigm or metaphor through which the behaviour of a program can be understood and related to the problem domain. Pascal and Ada are based on a procedural paradigm, where a program is organised as a hierarchy of procedures whose activation reflects a stepwise decomposition of the tasks needed to achieve a particular goal. Object-oriented programming originated in work aimed at simulating real systems; the first object-oriented language, Simula [9], introduced features that allowed the program to be organised in terms of the 'classes' of object that were being simulated. The development of Smalltalk [35] took this idea and turned it into a more general view of 'programming as simulation', that applied to all kinds of programming tasks, not necessarily those with system simulation as the explicit aim. In this model, a program is described in terms of how abstractions called *objects* interact. An object consists of a set of variables that represent its state, and a set of operations that define the services, or interface, that each object offers to the other objects within the system. Starting from its initial state, an object evolves as it responds to requests (*messages*) from other objects. To respond to a message, an object may in turn send messages to objects that it has access to.

### 3.2.1 PREMO Objects and Object Types

A PREMO object consists of a local (internal) state, and an interface consisting of a set of operations that can be invoked on the object. The state consists of a collection of variables, while the interface is a collection of operations (sometimes called *methods* in the wider literature). Conceptually, the state of an object cannot be accessed directly (read or modified) by other entities within the system. Instead, all access is mediated by the interface – a client can only access or modify an object's state by invoking one of the operations available in that object's interface.

Any object in a PREMO system may conceptually be active, that is, have its own thread of control. In practice of course, it is useful to only implement an object as a self-contained thread when this is needed; for many objects, for example those representing static data, there is no thread of activity, and therefore the implementation overhead can and should be avoided. This requirement does make it more complex to provide an implementation of PREMO in languages such as C++ [79] for which there is no standardised thread model. However, the notion of active objects is widespread in multimedia (see for example [65]. Adopting it as a design principle for PREMO has brought rewards, for example PREMO Part 3 defines virtual 'devices' which are both objects and processes.

Each object in a PREMO application is created as an *instance* of an *object type.* Object types correspond to the concept of a class in many object-oriented programming languages. The term 'object type' is used in the standard to make explicit the distinction between the *specification* of an entity in the PREMO standard, and the *implementation* of that entity in a particular programming system. There need not be a one-to-one correspondence between object types described in the PREMO standard and the classes in an object-oriented implementation of PREMO. In principle at least, it is possible to implement PREMO in a non object-oriented language. For similar reasons, the standard uses the neutral terms 'operation' and 'invocation' to refer to the behavioural component of objects, in place of the 'method' and 'message passing' which are biased towards object-oriented implementations. An object type introduces variables that will be part of the state of each instance of that type, and similarly, operations that will be available in the interface of each instance[1]. PREMO, like mainstream object-oriented languages such as C++, Ada'95 [7], and Java [36] has adopted a static object model, where the structure of an object is fixed by its type, and cannot be modified at run-time. Models have been proposed in which the structure of an object can be modified dynamically, i.e. while the system is running. Approaches such as CLOS [11] and Smalltalk [32], that include some capability for reflection', are good examples of this. A restricted dynamic object model is also found in the Python language [87], where an object can gain new operations. Although there is some evidence that multimedia systems might benefit from the use of a more dynamic model, the practical difficulties of realising this kind of behaviour in the languages currently in widespread use meant that the PREMO committee opted for a more traditional, static model. However, some support for dynamic structures was subsequently provided through the properties mechanism, discussed in Chapter 5.

### 3.2.2 Attributes

The restriction that all access to object state be via operations is strong, and in a number of cases imposes a significant overhead. For example, a device for playing media may have a number of parameters, such as speed of play, where it is intended that clients are free to inspect and modify these parameters as needed. Although the value of these parameters will affect the behaviour of the object, the operation of setting or accessing the parameter does no further work in itself. If the strict encapsulation model is followed in detail, each such parameter in an object type must be defined implicitly through a pair of operations for accessing and modifying the parameter. Doing this throughout the standard would have added considerable overhead to the specification of the object types. To overcome this, object types in PREMO can introduce *attributes*, which can further specialised into read-only, write-only, and read-and-write (the default). A read-only attribute can be seen as a shorthand for an operation that retrieves and returns the value of an internal state variable. Dually, a write-only attribute defines an operation that sets the value of a corresponding state variable. The default, an attribute that is both readable and writable, has both operations.

[1] The possibility of subtyping (see section 3.4) means that instances may also contain variables and operations inherited from supertypes.

For read-and-write attributes, an implementation of a PREMO object type may be able to realise the attribute simply as a variable that is publicly accessible, e.g. a 'public' state variable in C++ or Java. However, it should be noted that setting the value of an attribute can result in an exception, and to implement this it may be necessary to realise the attribute by a pair of methods. In any case, few languages support read/write permissions for variables, and consequently most attributes in the PREMO standard will map on to methods in an implementation.

### 3.2.3 Non-object Types

The ideal of object oriented programming, that every entity in the system is an object that can be manipulated through its interface, has probably found its purest expression in systems such as Smalltalk and SELF [83]. In this environment, all entities – even things such as numbers and characters – are construed to be objects. This brings a great simplicity to the language, but creates a significant difficulty. All data manipulation must (at least conceptually) be carried out by invoking operations, even things such as adding one number to another. This overhead can be reduced with good optimisation technology, but the resulting performance is still some way from that obtained with more direct methods of storage and operation. Partly for this reason, object-oriented languages have either been defined by adding features to existing programming languages (as in C++ and Ada'95), or by defining new languages that contain traditional data types in addition to objects (for example, Eiffel [66]and Java). Efficiency concerns aside, this approach also benefits from easing the migration path for existing programmers.

For very much the same pragmatic reasons, PREMO from the outset makes a distinction between object types, and non-object types. Each non-object *value* is a member of some non-object type, but non-object types are not part of the object type hierarchy described in section 3.5. Values of non-object types are atomic, there is no concept of a state or operations, rather, it is assumed that the environment in which PREMO operates provides a set of operators for manipulating non-objects, for example arithmetic and relational operators for working with integers and other numeric formats. There is however an important bridge between the world of objects and the world of values, which is discussed in the next section.

Part 2 of PREMO (see Chapter 5) introduces a collection of non-object types that are used widely in the remainder of the standard. Other PREMO components can and do introduce further, specialised types (in the form of subranges or enumerations).

### 3.2.4 Object Identity and Object References

A basic tenet of the object-oriented metaphor is that each object has an identity that persists, independent of the changing state [13]. Many object-oriented programming languages treat objects as pointer variables, with the object itself being but a pointer to where the state variables, and data needed for method dispatch, are stored. In this approach the identity of an object is implicit; it is the address of the state in memory. Some languages, notably C++ and Eiffel, also allow objects to be stored essentially ‘as values’, and in this case it is up to the programmer to take responsibility for defining any notion of object identity.

The PREMO standard mandates that all access to a PREMO object must be through an *object reference*; it is simply not possible to access an object directly in PREMO. There are two main reasons for requiring this:

1. PREMO objects are potentially distributed; if an object exists on a remote machine, it simply is not possible to access the object directly. By requiring that all objects be accessed via references, the standard allows a seamless migration from local to remote access, and avoids the complication of an explicit distinction between local and remote objects. An object reference can be used to encode both local address information and the location of a particular object across a network.
2. If PREMO were implemented in a non object-oriented language, it might be necessary to code a notion of object reference explicitly. Making the concept explicit in the standard provides at least a conceptual handle on the problem.

A PREMO client obtains a reference to an object by requesting a facility called an “object factory” (discussed in Chapter 5) to generate an object satisfying specific criteria. If it is able to satisfy the criteria, the factory will return a reference for the new object. Such a reference is a value of a **non object** data type, called an object reference, and can be used as any other item of non-object data. In particular, it can be copied, stored, passed as a parameter to operations, and compared for equality against other values of the type. It does not make sense to apply other comparison operators to object references. All subsequent activities involving the new object are then done via the reference. Consequently, the PREMO environment must provide support for activities, such as invoking an operation on an object, which are often taken for granted as part of a language-specific object model. In the case of a remote object, for example, an operation invocation must be translated into an appropriate remote invocation mechanism. Such assumptions have a significant impact on the binding of the *PREMO* specification to a implementation model; more will be said about this problem later in this chapter.

As a way of highlighting the important of object references within the standard, a notational convention was introduced: for object type *T*, ‘*ref T*’ denotes the non object type containing references to objects that are instances of *T*. In a language such as C++ that has explicit pointers, ‘*ref T*’ will become the type of pointers to *T*; for a language like Java in which object references are implicit, the declaration of an object of some class is understood implicitly as the introduction of a value of ‘*ref T*’.

In common with most object-oriented systems, PREMO introduces a special constant to designate an object reference that refers to no object. In the PREMO standard, this value is denoted *NULLObject*; in a Java implementation, it would be represented by the `null` object.

## 3.3 Operations

Object oriented programming languages often use the term ‘method’ for the definition of the process that objects of a particular class should perform in response to a request from some entity in the system. In keeping with the goal of separating the description of PREMO from language concerns, the term *operation* is used in the standard. An operation describes a process that can be applied to an object, through an *operation invocation* (also called an operation *request*).

Each operation has a signature, consisting of one or more input parameters, and zero or more output parameters, e.g. a signature has the form

$$op : p_1 : T_1, p_2 : T_2, ..., p_n : T_n \quad \rightarrow \quad r_1 : S_1, r_2 : S_2, ..., r_m : S_m$$

Here $p_1 ... p_n$ are the input parameters, and $r_1 ... r_m$ are the output parameters. The first input parameter to an operation represents the object 'on which' the operation is being invoked. In many object oriented programming languages, the 'receiver' of a operation invocation, or message, is implicit when an operation is defined, i.e. it is an object of the class in which the method is written. This object is then designated specially in the syntax for operation invocation, e.g. in Java or C++ the object on which the operation is applied is written before the operation:

$$receiver.opName(\ arg_1, arg_2, ..., arg_n\ )$$

There is a good reason for distinguishing the receiver. In the presence of subtypes (discussed in section 3.4), the operation performed in response to an invocation is determined by the type of the object that receives the request; this is called single dispatching. However, there are also languages which have a more general model of operations, for example Ada'95 and CLOS treat all arguments to an operation equally, and choose a suitable operation based on the types of *each* argument, so called multiple dispatching. The model of operations in PREMO is intended to be neutral with respect to this issue of language design.

The input and output parameters of a PREMO operation are required to be of non-object types, i.e. it is not possible to pass an object directly to an operation; instead, a reference to the object must be passed. If the object is of type *T*, then the parameter type will be of the reference type *Ref T*, which was previously mentioned, isa non-object type. This requirement on parameter types is another consequence of the need to distinguish carefully between objects and object references within a potentially distributed environment. By default, object references are passed directly to the called operation. It is possible to indicate that the operation should instead be passed a reference to a copy of the object, constructed using the create facility described in section 3.10.

## 3.4 Subtyping

Object oriented programs typically involve the manipulation of objects that have certain similarities. For example, a drawing program may manipulate objects representing points in 2, 3, or 4-dimensional cartesian coordinate systems. Each of these objects will have at least an x and y coordinate. A video media stream may pass from a device which reads data from an external source to a device that displays the video; while the devices may be objects of different object types, both however may have the capability to stop, pause, and resume their processing tasks. Subtyping is the relationship that holds between two object types when objects of one type can be used in contexts where objects of the first type are 'normally' expected. In the examples above, a *3Dpoint* object type may be a subtype of *2Dpoint*, while both the video stream producer and render may be subtypes of a generic object type representing video devices. The definition of a PREMO object type should indicate which object type(s) it is a subtype of.

Exactly what should and should not constitute 'a subtype' has been the subject of much debate within the object-oriented systems communities (see for example [88]), and indeed different approaches can be required at various stages of the software development process. PREMO bases its notion of subtype on the interface of the object types. For $T$ to be a subtype of $S$, the following conditions must hold:

1. for each operation $OP_S$ in $S$, there should be an operation $OP_T$ in $T$ with the same name;
2. the number of parameters accepted by $OP_S$ and $OP_T$ should be the same;
3. with the exception of the first parameter, each the type of each parameter to $OP_T$ should be the same as the type of the corresponding parameter to $OP_S$. The exception is because the first parameter to an operation is taken to be a reference to the object on which the operation is dispatched. Therefore this type will by definition be distinct in different object types.

Two concepts which are useful in discussing subtyping are direct instance, and immediate subtype:

- Each object in PREMO is a direct instance of exactly one type, which is called the object's immediate type. If $O$ is a direct instance of $T$, this means that $O$ is an instance of $T$, but not an instance of any subtype of $T$. Intuitively, an object is a direct instance of the object type used to create the object.
- An object type $T$ is an immediate subtype of object type $S$, if, informally, $T$ is 'immediately beneath' $S$ in the object type hierarchy. More precisely, $T$ must be a subtype of $S$, and there must be no other object type $U$ such that $U$ is a subtype of $S$, and $T$ is a subtype of $U$.

An object type can have any number of subtypes, and can also have any number of supertypes, that is, it can be a subtype of multiple object types. An object type, plus the collection of its supertypes, plus their supertypes, etc., forms a directed graph called the type graph of the object type. New subtypes are created by subtyping from some existing PREMO object type. In particular, all PREMO objects are subtypes (directly or indirectly) of the type PREMOObject, which defines the minimal functionality of each object in a PREMO system. This includes:

- operations to enquire an object's type and type graph;
- an initialize operation that is performed on an instance when it is first created; and
- an `initializeOnCopy` operation to be performed on an instance if it is created by copying an existing instance.

Some object types in PREMO have been introduced, not because they define objects that are useful in themselves, but because they describe some significant functionality. Such an object type is called an *abstract type*. The functionality described in an abstract

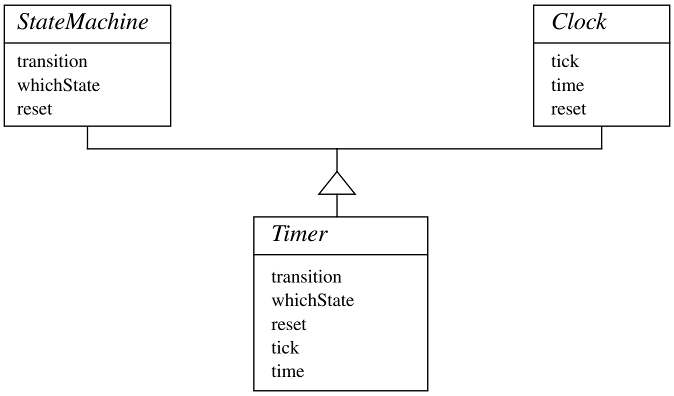


Figure 3-1 — Name Collision in Multiple Inheritance

type might be realised in a number of subtypes. By subtyping from the abstract type, these object types then clearly indicate that they are providing particular services. Abstract types are non-instantiable, i.e. it is not possible to have an object which has an abstract type as its immediate type. Instead, any behaviour of an abstract type must be accessed through instances of some subtype.

## 3.5 Inheritance

Where subtyping is a *relationship* between object types, inheritance is a *mechanism* for re-using the definitions of one object type in the description of another. Informally, if object type *T* inherits from *S*, *T* acquires all of the state components and operations of *S*; it may also introduce state and operations of its own. If a PREMO object type *T* inherits from an object type *S*, *T* is defined to be a subtype of *S*.

Multiple inheritance, where an object type can inherit from more than one other object type, is supported in the PREMO object model. While it is useful to be able to combine features from various object types, in practice multiple inheritance introduces the potential for conflict between names (of states or operations) inherited from different sources. For example, Figure 3-1 shows a *Timer* object type being defined by multiple inheritance from object types representing a *StateMachine*, and a *Clock*. Both the *StateMachine* and *Clock* introduce an operation called *reset*. The problem is that the Timer object type inherits two conflicting implementations of *reset*.

Programming languages have adopted various approaches to dealing with this problem. C++ for example a number of rules that attempt to resolve potential ambiguities. Java, in contrast, does not allow multiple inheritance of implementations, but does allow a class to implement multiple interfaces. Because an interface is only an operation signature, there is no scope for ambiguity. Multiple inheritance was felt to be a useful technique for deriving new object types in PREMO, and has been used in a number of places in the standard. However, it was felt that the definition of rules to resolve any ambiguity would impose an unnecessary implementation bias in the standard. Instead, features (state variables, attributes and operations) of an inherited object type can be renamed in order to avoid clashing with similarly named features from other object types that are also inherited. Inherited features can also be redefined; for example, the *Timer* object type in Figure 3-1 might define a new *reset* method that overrides both inherited versions. If a name clash does occur, the results are not defined by the PREMO standard.

## 3.6 Protected Operations

Certain operations available on a PREMO object are not intended for use by clients. Two examples of these are the `initialize` and `initializeOnCopy` operations defined in PREMOObject and are thus available in all objects. These are intended for use by specialised facilities described in section 3.10, not by arbitrary clients. Access to operations can be limited by the declaring the operation as *protected*. A protected operation can only be invoked by an instance that contains the operation. The behaviour of a protected operation can be modified within subtypes, for example by overriding, but an operation declared as protected cannot lose that status in a subtype. Attributes can also be declared as protected, meaning that the operations for reading and/or writing the corresponding state component are marked as protected.

## 3.7 Operation Selection, and Casting

Operation invocation in object oriented systems is complicated by the presence of the subtyping relationship. When an operation is invoked, its name is given, along with (non-object) values for each input parameter. The type expected for each result parameter is also known. On the basis of this information, the PREMO environment must select which of a number of possibly matching operations is to be executed. It does this by examining the immediate type of the controlling parameter (the first input argument, which is always present). The following conditions must be met by any operation that is a candidate for execution in response to the request:

1. The immediate type of the object must define an operation with the same name and number of input and output parameters as appear in the invocation. Suppose that the signature of this operation is

   $$op : p_1 : T_1, p_2 : T_2, ..., p_n : T_n \quad \rightarrow \quad r_1 : S_1, r_2 : S_2, ..., r_m : S_m \,;$$

   two further conditions must then be met.

2. For each input parameter $p_i$,
   - if $T_i$ is *ref T*, for some object type *T*, then the corresponding actual parameter value must be a reference to an object of type *T*, *or an subtype of T.*
   - if $T_i$ is not a reference type, then the corresponding actual parameter must be a legitimate value of type $T_i$.

3. For each output parameter $r_i$,
   - if $S_i$ is *ref T*, for some object type *T*, then the destination for the corresponding result must be of type *ref U*, where *U* is either *T* itself, *or some supertype of T.*

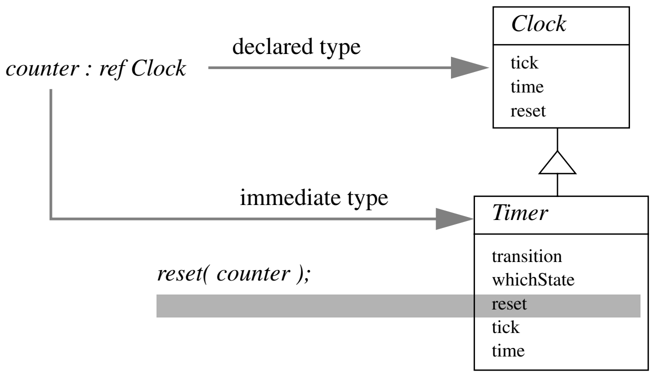


Figure 3-2 — Operation Selection

- if $S_i$ is not a reference type, then the destination for the corresponding result must be of type $S_i$.

Points 2 and 3 can be summarised by saying that PREMO assumes a contra-variant rule for operation parameters.

If it happens that more than one operation in the immediate type of the controlling parameter meets requirements 1-3. If the signatures of the operations are the same, the PREMO standard requires that an exception be raised (see section 3.9). If however the operations have different signatures which are nether the less compatible with the parameters given by operation invocation, the environment of PREMO may define a rule or mechanism for choosing one of the candidate operations to be executed.

The importance of selecting an operation based on the immediate type of the controlling parameter is shown in Figure 3-2, using the object types from Figure 3-1. Here, an object reference, *counter*, declared as type *Ref Clock* has been assigned a *Timer* object. This is quite legitimate, since a *Timer* is a *Clock*. However, if the *reset* operation is now invoked on *counter*, the operation that is executed should be the version of *reset* defined by the *Timer* object type, not that defined by the *Clock* object type. And indeed this is what will happen, since the immediate type of *counter* is *Timer*, rather than *Clock*.

A PREMO environment is required to provide a finer level of control over operation selection via a `cast` facility. If the program containing the counter reference shown in Figure 3-2 needs to access the *reset* behaviour defined in the *Clock* object type, it can use the facility to generate a new object reference which has *Clock* as its immediate type. The required behaviour can be then be accessed by invoking *reset* using this new reference as the controlling parameter. An object reference can only be cast to a supertype of its immediate type, i.e. an object type that appears in the type graph of the immediate type. This is supported by facilities defined in Part 2 of PREMO that allow all objects to enquire their type graph.

## 3.8 Operation Request Modes

In simple models of program execution, operations are usually carried out as synchronous processes, with the caller suspending until the operation has been completed and the result (if any) is returned. This model of course breaks down in the presence of concurrent processes, and is untenable in a distributed system where the overhead of locating and communicating with a remote object can be non-trivial compared with actual execution times. Distribution has been a fundamental design goal in PREMO, and consequently the PREMO object model must address the problem of how operation invocation takes place. In PREMO, each operation on an object is defined to operate in one of three possible request modes. These modes are called synchronous, asynchronous, and sampled. The mode is an immutable property of an operation, specified when the operation is defined. A subtype can override the implementation of an operation, but cannot change the mode of the operation.

- *Asynchronous* operation request causes the caller of the request to be suspended until the request has been serviced and a result returned. This is the usual model found in and supported by default in most object oriented programming languages.
- The caller of an *asynchronous* request can continue its own thread of control as soon as the call is made; at some point the requested operation will be invoked. By the time the operation has been completed, the caller may be carrying out some different task, and it is not therefore possible to return a result. Communication between asynchronous processes is a well known engineering problem, involving techniques such as shared variables with mutual exclusive access.
- A *sampled* request is similar to an asynchronous request, except that any pending request for a given operation (i.e. a call that has not been serviced) is over-written by any new request. Conceptually, each operation has a 1-place buffer for storing pending requests. Sampled mode has been supported in PREMO for tasks where the rate at which an object can be informed of, and service, requests may be slower than the changes that are causing clients to make requests. This may happen for example where a server graphical objects in a distributed VR environment is being asked for detailed object geometry by clients managing the interaction of remote participants.

Some care is needed in dealing with the suspension that results from invoking a synchronous operation. When suspended, an object can still receive requests from other objects, which are managed in accordance with the behaviour described above. For example, a synchronous request on a suspended object will be held as pending, with the caller itself suspended, until the callee's own invocation is completed and it is able to service the request. While suspended however, an object can continue with its own internal thread of processing; it just cannot access information related to its own receptors, for example to service a request. When an object invokes an operation on itself, the operation receptor mechanism is by-passed; the implementation of the operation is invoked immediately. If this were not done, such an invocation would deadlock.

Distribution, and the activity of objects, both mean that an object may receive operation requests concurrently. At any time, an object may have a number of requests outstanding on any number of its operation receptors. In this case, the object chooses one of these requests non-deterministically and services it, then selects another request. An object also has the ability to limit the range of requests from which it will select one. This facility is for example provided in the Ada programming language through the `select` statement.

## 3.9 Exceptions

An exception is a situation that arises during the execution of a PREMO operation which makes it impossible or inappropriate to continue the execution of the operation. For example, an object may be in a state for which the operation is inappropriate, or a parameter value may be illegal. If a PREMO operation detects that an exception condition has been broken, execution of the operation should be abandoned, and an exception should be raised. The state of the object is left unchanged (note that all exceptions defined in PREMO relate to conditions that can be checked without modifying the state of any object involved).

How an exception is raised, and communicated back to the object that invoked the operation, is not defined in PREMO. Different programming languages support more or less explicit constructs for exception management, and, for PREMO, exceptions are part of a larger issue called the environment binding which is discussed in section 3.11. PREMO however does mandate are the following rules.

- Different error conditions result in different exceptions, i.e. it is possible to determine the nature of the error that caused the exception from the exception itself. Exceptions are defined in the PREMO standard as objects that, in addition to their identity, can convey additional information about the cause of the exception.
- Facilities must be available for the caller of an operation to detect when an exception has occurred, and to unpack the information provided by exception objects.

One kind of operation is treated differently. Operations that initialise an object, described in the next section, do not themselves raise an exception, instead, the services of the PREMO environment responsible for object creation should raise the exception. If an exception occurs during object initialisation, the *NULLObject* reference is returned as the result of the creation operation.

## 3.10 The Object and Object Reference Lifecycle

PREMO objects are created, manipulated and destroyed by facilities that form part of what is called the environment binding. This binding will be discussed in general terms in the next section; here, we describe the changes can take place in the evolution of an object and object reference.

1. **Creation**. An object is created using the `create` facility from the PREMO environment. This facility is given the name of an object type of which an instance is required. It is required to produce a suitable instance, and to invoke the protected `initialize` operation on the instance. Any parameters for initialisation are passed via the create facility, which returns either an object reference, or the *NULLObject* reference if an exception occurred during initialization. In the former case, the immediate type of the reference will be the object type specified as an argument to the create facility.

2. **Copying**. An object can be copied using a `copy` facility, which is given a reference to the object to be copied. Like the `create` facility, `copy` returns a reference to a new object, satisfying the following requirements.

   – The reference returned by `copy` has the same immediate type as its argument.

   – A protected operation, called `initializeOnCopy`, is invoked on the new object by the copy facility. This operation does not accept any arguments (other than the controlling parameter, which is the new object itself). The `initialize` operation is not invoked by `copy`.

   – If the copy is *shallow*, each component of the state of the new instance is set to the value of the corresponding component in the original instance. In particular, this means that a reference to an object in the state of the original instance becomes a reference to the same object in the state of the new instance.

   – If the copy is *deep*, all non-object components of the state of the new instance are set up as copies of the corresponding components in the original instance. Each object reference in the new instance is then set to be a new reference produced by recursively invoking the deep copy operation on the corresponding reference from the original instance.

   The choice of shallow versus deep copy is indicated at the point where the `copy` facility is utilised.

3. **Destruction**. A reference to an object can, conceptually, be destroyed using a facility called `destroyReference`. The result of using this is that the reference will have the value *NULLObject*. The object, however, may still exist and be accessible via other references. The term ‘conceptually’ is used here, because the destruction of a reference is in many cases little more than the assignment of a (new) value to a variable representing an object reference. Object instances are destroyed similarly through the `destroyObject` facility. The effect of `destroyObject` on any reference to that object is not defined. It is good practice for the PREMO application to invoke `destroyReference` systematically, on each (former) reference to a destroyed object.

## 3.11 The Environment Binding

Facilities such as active objects and asynchronous operation dispatching impose requirements on an implementation of PREMO. How they are realised depends both on the programming language to which PREMO is bound, as well as the *environment* that the implementation uses. For example:

- a Java implementation may realize object activity through the Java threads mechanism, with access to remote objects provided by the RMI package [39];
- a C++ implementation might support active objects through a separate thread package (pthreads, for example), while remote object access may be based on an external system such as CORBA [71].

  In both cases (Java or C++), some facilities, such as operation request modes, may need to be programmed explicitly.

It is not in general sufficient to produce just a language binding for PREMO. An *environment binding* may also be needed, to define the link between certain services required by a PREMO system, and some higher-level mechanisms implemented on top of the selected programming language. There is also no fixed border between language and environment binding. Some programming languages may for example support distribution as a fundamental concept, and therefore services such as remote method invocation will be built in. Most languages however do not have such services, and instead remote method invocation will need to be realised through a binding to a separate set of higher level services, for example the Java RMI package, or a CORBA interface.

It has already been mentioned that PREMO assumes that its environment will provide a number of fundamental services, including:

- object creation and destruction;
- copying;
- operation invocation;
- generation and detection of exceptions

These facilities might be realised through explicit constructs in the host programming language, or, like remote method invocation, they might need to be supported by some package or other higher-level facility. To allow an implementor freedom to select the most appropriate tools, these facilities are viewed not as fundamental components of PREMO, but as services that a PREMO system requires from its environment. An implementation of PREMO must therefore include a mapping from these generic facilities onto appropriate implementations.

# Chapter 4

## General Implementation Issues

PREMO is an abstract specification, i.e., the ISO document describes its functionality in a programming language and environment independent way. This follows the traditions set up by various standards, including standards in computer graphics such as GKS or PHIGS[5]. Whereas this is perfectly appropriate for an official ISO Standard, we feel that, for the purposes of this book, it would be better to present PREMO in a less abstract manner. This means:

- specifying objects in a well–known programming language rather than an abstract specification formalism used in the ISO document; and
- referring to a real, albeit currently prototypical, implementation of PREMO.

To illustrate the first point, Figure 4-1 shows how an object is specified in the Standard text (you do not have to understand all the details for now). This specification uses a formalism inspired by Z[78] and Object–Z[24,25]; it is precise and usable, but it obviously requires a certain practice to read. By using a well–known language to define our objects, the reader will have less difficulties to understand the type specifications themselves, and can concentrate on the semantics of the object types rather than the formalism!

The prototypical implementation, referred to above, is being developed in parallel with writing this book. We will therefore refer to "real" objects, rather than just a set of paper–and–pencil entities; all the type specifications appearing in the book will undergo the scrutiny of a compiler, thereby ensuring the correctness of all the code examples in the book.[1] "Prototypical" means that the implementation will only concentrate on the key elements of a possible product–level implementation, but will ignore certain details which do not add to understanding the essence of the PREMO model. As an example, a powerful MPEG coder and decoder is obviously important in a multimedia system; however, our prototypical implementation does not aim at a very efficient implementation of the MPEG compression algorithms, it just shows where a powerful coder/decoder can be plugged in. Of course, the prototypical implementation can (and hopefully will) be used as the basis for a production level PREMO implementation, but this goes beyond the scope of this book.

[1] All the specification code of the book are also freely available on the web site:ftp://ftp.cwi.nl/pub/premo

$\mathit{PropertyInquiry}_{abstract}$

*EnhancedPREMOObject*

*inquireNativePropertyValue*

$\mathit{key}_{in} : \mathit{Key}$
$\mathit{nativeValue}_{out} : \mathrm{seq}\ \mathit{Value}$
$\mathit{exceptions} : \{\mathit{InvalidKey}\}$

The operation returns the native property values for $\mathit{key}_{in}$. This native property value represents the value or values the project instance can take on for $\mathit{key}_{in}$. A native property value is available for all properties which are defined as part of the object's functional specification.

The $\mathit{nativeValue}_{out}$ is typically a sequence of values (e.g., if the type of the corresponding property is defined as a *String*), or a minimum–maximum range (if the value is a numerical type). The specification of the property shall define the return type of the result of operation invocation if this is not the case.

Exceptions raised:

| | |
|---|---|
| *InvalidKey* | The key is invalid. |

*PropertyInquiry*

Figure 4-1 — Object Specification in the ISO PREMO Document

The core of this book describes the object interfaces and the semantic behaviour of the objects only, to give a thorough presentation of the reference model advocated by PREMO. However, a separate appendix gives also some more details of the implementation itself, for those who want to understand what happens "behind the scenes", and who may feel challenged to provide a full implementation of PREMO.

The choice of the programming language and the environment used for the implementation has a consequence on the way objects will be presented throughout this book. The current chapter concentrates on some of the general issues and constraints which directed our choices and how these constraints will be visible in the various type specification in the rest of this book. Of course, the problems we will describe, and the answers we have found to those problems, reflect the chosen environment as well as our own software engineering abilities, and the reader might perfectly be capable of greatly improving our design. But that is all right; the goal of this book is to understand the reference model of PREMO, and an efficient implementation would not necessarily be the most direct and clear account of PREMO.

When engaging into the implementation of any abstract specification, one has to choose the programming language as well as the programming environment where this implementation will exist. This is what we will detail in what follows.

## 4.1 Implementation Choices

### 4.1.1 Implementation Language

PREMO is an object–oriented standard. This means that the functionalities of PREMO are defined in terms of object types and their behaviour. It is therefore quite natural to choose an object–oriented programming language for the implementation.

However, things are not that simple. Traditional imperative languages, such as Fortran or C, rely on a common computational model which can be broadly characterized as a "von Neumann machine", i.e., they all use concepts such as common variables, procedures, input and output variables, etc. Although there are of course great differences among these languages, the underlying model is the same. As a consequence, the mapping of standards such as, in the area of computer graphics, GKS or PHIGS (see, e.g., [5]) is relatively straightforward; these standards are defined as a large set of abstract functions, which have to be mapped against the specific features of a particular imperative language. There are a lot of details to fill in, but it is straightforward.

In spite of the abundance of object–oriented languages, the situation is far from being that simple when an object–oriented design is adopted. One has to realize that the term "object–orientedness" is, though widely used, rather vague, and each author of a particular object–oriented specification or each language designer have his or her own view on how objects are defined and how they behave. To be somewhat more precise, the *object models* used in these designs differ. These differences are then reflected in the various programming languages that claim to be object–oriented (Smalltalk[31], Eiffel[66], C++[79], Java[36], Python[87], Ada'95[7], to name only a few).

The differences can be very significant when it comes to an implementation. Without trying to be exhaustive, here are some characteristics which may be different from one object model to the other (see also [13]):

- single vs. multiple inheritance (e.g., Java has single inheritance, C++ has multiple); indeed, is the object model based on inheritance at all, or does it rely on the concepts of prototypes and delegation (see, e.g., [62]);
- separation, or not, between an interface and an object type (e.g., Java supports the separation explicitly, Smalltalk, Python, or C++ do not[1)]);
- existence of non–object types, such as integers and doubles (e.g., Smalltalk has only objects, C++ and Java have non–object types, too);
- how objects are created, destroyed;

[1)] Although one could argue, in the case of C++, that abstract C++ classes may be considered as interfaces.

- are the interfaces of object instances fixed (e.g., is it possible to add new attributes or methods to an object instance such as, for example, in Python or a delegation based model);

and the list continues…

As a consequence of this diversity, if a very precise abstract specification is to be defined, this specification must include its own object model description. This has been the case for such industrial specification as CORBA[71]: OMG has its own specification of what objects are in OMG's point of view. And this is the case with PREMO, too; an important portion of the so–called "fundamental" part of PREMO (see Chapter 3) has been devoted to the definition of the object model which PREMO relies on. Although this object model is not particularly different from what people usually think of objects, it had to be precisely defined, and this model did influence our choice for a programming language as far as our implementation was concerned.

The language we have finally chosen is Java. Although the object model of Java is not 100% identical to PREMO's model, it is probably the closest we can get. Apart from its large popularity and wide availability, the following features of Java influenced our choice:

- Java has exceptions as part of the language (exceptions are also used in PREMO);
- threads are integral part of the Java design;
- Java has a large and very useful set of utilities in terms of the various Java packages;
- good portability across numerous platforms, including both the language and the set of 'core' Java packages.

As a consequence, all object specifications, example code, etc., in this book will be in Java[1)].

Java uses the concept of 'packages' which offers us a nice way of structuring the implementation, as well as the various interfaces of PREMO. Our implementation is split into the following packages:

```
premo.std.part2
premo.std.part3
premo.std.part4
premo.impl.part1
premo.impl.part2
premo.impl.part3
premo.impl.part4
premo.impl.utils
```

The content of these packages is straightforward: the '`std`' packages contain those classes and interfaces which are the direct counterparts of PREMO specifications, the '`impl`' packages contain the various implementation specific classes and interfaces. The `premo.impl.utils` package separates those implementation classes which are used in all parts of PREMO; the `premo.impl.part1`, etc., packages contain the implementa-

[1)] We will rely on the familiarity of the reader with Java. Apart from the book on Java by the designers of the language[36], there are a large numbers of other books available (e.g., [28] or [39], to refer only to those used by the authors), and the reader may consult these if necessary.

tion classes which are specific to a PREMO part. Part 1 of PREMO does not include object type specification, i.e., it does not appear in this list as a `std` package; however, some of the requirements of the object model lead to the development of special facilities, which constitute the `premo.impl.part1` package.

### 4.1.2 Implementation Environment

Having an implementation language is not enough; a full implementation environment has to be chosen, too. This entails the usual utility and I/O facilities, basic data structures such as hashtables, vectors, etc. Java offers a standard set of packages (the "java core") which are available with all Java implementations, and which are perfectly appropriate for our implementation (e.g., the `java.lang`, the `java.utils`, and the `java.io` packages). There are no real problems in this respect.

There is, however, one area where a further choice has to be made, and this is distribution. Indeed, one of the main characteristics of PREMO is that it should function in a fully distributed environment, i.e., some of the PREMO objects can be accessed and invoked through a network. This means that, beyond the choice of Java, we also have to choose which kind of tool we would use to manage distribution. It also turns out that his choice has its (albeit minor) consequences on the way the various specifications in the book are defined and presented.

At the time of writing this book, there are, broadly speaking, three alternatives one could choose from to control the distributed access of Java objects (see, e.g., [70] or [86] for further details):

4. Microsoft's DCOM (Distributed Component Object Model) used with Java (i.e., Visual J++);
5. Some form of an OMG CORBA implementation with a Java interface (ILU, Netscape's ONE or Caffeine, OrbixWeb, etc.); or
6. Sun's (i.e., Javasoft's) own RMI (Remote Method Invocation) facilities, bundled into Sun's JDK (Java Development Kit), starting from version 1.1.

We ruled out the usage of DCOM for portability reasons; we did not want to have an implementation dependent on only one environment. The choice between RMI and a CORBA implementation is less obvious. CORBA offers a greater compatibility with other object environments and programming languages, and a richer set of functionality with respect to object interface registry (although, by the time the reader reads these lines, Sun may come up with additional registry services for RMI which are compatible with CORBA). On the other hand, using CORBA means having to access and use yet another large piece of software besides the Java environment itself, which may prove to be a significant burden for portability (not all CORBA implementations are available on all platforms where Java runs). We have therefore opted for RMI. Provided we use Java JDK 1.1 or higher, this provides us with the level of portability Java itself has; it also has the functionality necessary for our prototypical implementation. It must be emphasized, though, that a fully marketable implementation of PREMO might well prefer to choose CORBA rather than RMI; development in this area is so rapid that our choices may have been different had they occurred at a later time. Actually, the design philosophies behind CORBA and RMI are very close to one another, so most of the specification and code in this book would be valid for a CORBA environment, too. Both environments rely on the concepts of client and server stubs, and the major differences between the two approaches can be localized in the way these stubs (i.e., remote objects) are created, and the references to the server objects are accessed. As we will see in Chapter 5, the abstract PREMO specification localizes these functionalities into one or two objects anyway, so the differences between the RMI and corresponding CORBA implementation are minor and easily manageable.

There is one important difference, however. In the case of CORBA, *all* objects in the CORBA world are, per definition, remotely accessible, hence only their references should and can be passed as method arguments. Java's RMI, on the other hand, allows the user to differentiate between objects which are remotely accessible (i.e., they 'register' as remote server, have a server and client stubs) and objects which cannot be accessed through the RMI mechanism but still exist in their own right. When transferring references to objects in the second category, RMI passes the objects by value[91].

The possibility of passing objects by value is important from PREMO's point of view. It so happens that the PREMO specification also differentiates between objects which have their services available through the network, e.g., full–blown multimedia players, and objects which are short–lived, simple, and are not supposed to provide complicated *services* in a distributed environment, e.g., a geometric point (see section 5.3.4). The distinction made by the RMI designers between 'remote' and 'local' objects perfectly suits our needs.[1)]

## 4.2 PREMO Specifications in Java and Java RMI

### 4.2.1 Constraints on the Specification Details

Using RMI for the specification of potentially remote objects creates some constraints as to how the abstract PREMO object type specifications should be described in Java terms. First of all, in Java's RMI, stubs are created for *interfaces* and not for classes. In other words, one has to define a remote interface, which is used by the `rmic` stub generator program to create the client stub and the implementation skeleton. The real service itself is supposed to be defined through a separate remote server class which implements the remote interface. This means, in practice, that the public interface of all PREMO objects which are remotely accessible (a more precise definition will be given in Chapter 5) will be defined as Java *interfaces* and not as Java *classes*. These interfaces will be placed into the '`std`' packages. The implementation has to provide classes which implement these interfaces; these will form the '`impl`' packages.

1) Passing objects by value is not yet defined in CORBA, but OMG has issued a Request for Proposal in 1996, and a specification may become available by the time this book goes to press. So, on long term, this difference may disappear, too.

Another consequence of using RMI is the requirement that *all* methods in a remote interface are supposed to throw the `java.rmi.RemoteException`. The fact of throwing those exception also means that all method invocations should be surrounded by `try` and `catch` clauses.[1] We have decided *not* to include the `throw` clause for `RemoteException` in the text describing the interface and the semantics of PREMO objects, nor do we enclose these method invocations in a `try–catch` pair in the example code fragments in the text. Our concern was clarity and readability rather than minute details. The detailed Java specification of all PREMO objects appear separately in Chapter 8; those specification are fully precise and include the relevant `throw` clauses, too.

Let's look at a simple example. PREMO defines an object called `Clock`, and an object called `SysClock`; the latter is a subtype of the `Clock` object. The `SysClock` object is defined as a Java interface as follows:

```
package premo.std.part2;
public interface SysClock extends Clock {
   /**
   * Returns the number of tick since the start of the
   * PREMO era, i.e., 00:00am, 1st of January 1970, UTC.
   */
   long inquireTick() throws java.rmi.RemoteException;
}
```

The interface extends the interface `Clock` which, indirectly, extends a higher level object in the PREMO hierarchy which, on its turn, extends `java.rmi.Remote`. In other words, the interface is (indirectly) defined to be potentially remote.

The following implementation class corresponds to the `SysClock` interface:

```
package premo.impl.part2;
import premo.std.part2;
public class SysClock_Impl extends Clock_Impl implements SysClock {
      public long inquireTick() throws java.rmi.RemoteException
      {
         ...
      }
}
```

We omit the implementation details here, as well as the details on the constructors, initialization, etc.

Note that the class `SysClock_Impl` is a subtype ("extends", in Java–speak) of the class `Clock_Impl`. This reflects the inheritance relationship defined in PREMO. This scheme cannot be followed in all cases: Java does not allow for multiple inheritance of classes, only of interfaces, whereas PREMO does allow for multiple inheritance. Fortunately, the usage of multiple inheritance is rather limited in PREMO, so the scheme used for `SysClock` can be considered as fairly typical.

Finally, a fully functional Java code fragment would use the object as follows:

[1] Note that if a CORBA implementation were used, the constraints would not be very different. For example, if the objects are defined in Netscape's Caffeine, the only difference is that the exception to be thrown is called `CORBA.SystemException` instead of `java.rmi.RemoteException`. Just as in the case of RMI, objects should be defined in terms of interfaces for Caffeine, too.

```
SysClock sysClock;
sysClock = (SysClock)GetTheObjectReferenceFromSomwhere();
int time;
try {
   time = sysClock.inquireTick();
} catch( java.rmi.RemoteException e ) {
   System.out.println(e.toString());
}
```

although, as we said, we will not always include the `try` and `catch` clauses into all our examples. The `GetTheObjectReferenceFromSomwhere` is obviously a dummy method call for now; what should be noted, though, is that the method could return either a reference to a local object or a reference to a local stub; the code remains identical.

Average users of PREMO may refer exclusively to the interfaces defined in the '`std`' packages. By the very nature of Java interfaces, these describe the publicly accessible methods. However, PREMO also defines some protected methods, i.e., which are of importance for subclasses only. These are of interest for programmers who wish to extend existing objects. These protected methods are included in the '`_Impl`' classes; indeed, extending existing PREMO object types means in practice that these classes should be extended in the Java sense.

As said before, this "dual" structure, i.e., the separation of a separate interface and its implementation, is valid for those PREMO objects which may be used as server objects through RMI. PREMO also includes some simpler objects, which are implemented in a more "straightforward" manner; more about it in the coming chapters. Also, the choice of Java and Java's RMI have some other, minor consequences on the way specific abstract PREMO specifications are mapped onto an implementation. These consequences may be better understood if the specific PREMO concepts are also known, so we will have to come back to those in later chapters.

### 4.2.2 Registering Server Objects

An issue which has not been addressed up to now is how an object instance, once created, becomes a server object, i.e., how would its methods become available for remote method invocation.

The usual approach when using RMI is that the implementation class would extend from a `java.rmi.RemoteServer` class; in the current release of JDK this can be done by extending it from `java.rmi.UnicastRemoteObject`. By ensuring that the constructor of the superclass is invoked at instantiation time, such an object instance would then automatically be registered as a server object. We say "usual", because most of the Java textbooks, when describing RMI, describe this approach only.

This scheme, however, would lead to problems with the PREMO implementation classes. Indeed, Java does not allow for multiple inheritance; i.e., if a server class would already have to extend `java.rmi.UnicastRemoteObject` in order to become a server object, the code of, for example, `SysClock_Impl` above would become invalid. Furthermore, this would mean that *all* instances of the given object type would become server objects which, though semantically acceptable, may have some negative consequences on efficiency.

Fortunately, the designers of RMI anticipated this problem. An object can also be registered as a server object by issuing the call:

```
UnicastRemoteObject.exportObject(InstanceRef);
```

call. This can be done either by the object instance itself, or by any other object. This is the approach we have taken in our PREMO implementation; the object is registered (“exported”) by the so–called factory objects; more about this in section 5.7.3 on page 123. That is also why there is no trace of registration in the example code above.

# Chapter 5

# The Foundation Component

## 5.1 Introduction

The foundation component of PREMO provides a number of object types and non–object types which are used in all the other PREMO components. These are rarely used in isolation, but rather as constituents of more complex PREMO objects. This component also defines the top–level of the full PREMO type hierarchy, thereby ensuring a set of common facilities which all PREMO objects have.

The reader will realize that the terms "multimedia", or "media", will be very rarely used when describing the objects in the foundation component (except for the synchronizable objects). This may sound strange at first glance, but this is a necessity. The goal of the foundation component is to provide those fundamental building blocks which are necessary to do real multimedia processing and not to define the real processing units themselves. These are left for further components.

## 5.2 PREMO Non–object Types

Similarly to a number of object–oriented programming languages and programming environments, the PREMO specification includes so–called non–object types. The reason is primarily efficiency, i.e., to have the ability to define integers, floats, etc., without paying for the overhead of object creation, object references, and so forth. (Actually, with a few exceptions, most object–oriented languages that are feasible PREMO targets would support this.)

PREMO does not really define what non–object types are, it rather tells us what they are *not*. Non–objects represent final entities, i.e., there is no notion of subtyping for non–objects. Non–objects are not part of the full PREMO object hierarchy (see section 5.3 below). Non–objects cannot offer services over the network, nor can they be active clients of those services. On the other hand, non–objects also have types, just as objects do. Also, non–objects, and *only* non–objects, can appear as input or return arguments for object methods. This last point seems to be a severe restriction at first glance, but it is not. Indeed, PREMO defines object references as being non–objects, too. What this restriction means is that only object *references* can appear as input or return arguments, and not the objects themselves.

PREMO non–objects fall into three categories. These are as follows.

1. *Basic data types*. This is defined as a small set of non–object data types describing fundamental entities such as integers or floats. All the basic data types are defined in Part 2 of PREMO.
2. *Constructed data types*. It is possible to construct new non–object types using existing ones. A characteristic example is the construction of arrays. All Parts of PREMO contain a small number of such constructed data types, related to the proper specification of "real" PREMO objects.
3. *Exceptions*. These are, strictly speaking, not defined as non–object types in the PREMO documents, but they logically belong to this category. Error management in PREMO is based upon the ability to throw exceptions from within a method, exceptions which can be then caught by the caller. Exceptions carry information which can be extracted by whoever catches them.

It may be a source of confusion that some of these entities, referred to as "non–objects" in PREMO, will be represented by Java classes, i.e., objects. This is the consequence of the nature of Java, which does not have structures, enumerations, etc., and encourages the programmers to use classes even for those relatively simple entities[1]. However, these "non–object classes" have some characteristics in common, which differentiate them from the Java classes representing "real" PREMO objects. Namely:

- They are all `final` classes, i.e., they cannot be extended by other classes.
- They are not part of the PREMO object hierarchy.
- They are never defined as possible server objects for RMI.
- They all implement the `java.io.Serializable` interface, i.e., they can be passed as arguments through RMI calls.
- They are not active entities (i.e., do not run in separate threads), whereas PREMO objects make extensive use of threads.

When necessary, we will refer to these classes as "non–object classes".

The subsequent sections will give some details on each of the non–object categories.

### 5.2.1 Basic Data Types

Most of the PREMO basic data types are fairly straightforward, and they have their direct counterpart in Java. These are:

| **PREMO non–object types** | **Java types** |
|---|---|
| Integer ($\mathbf{Z}$) | `int` |
| Real ($\mathbf{R}$) | `double` |

[1] More fundamentally, there will rarely be a direct mapping between the PREMO view of objects and non–objects, and that of any target programming language.

| PREMO non–object types | Java types |
|---|---|
| Object type | `java.lang.Class` |
| Time | `long` |
| Object references | implicitly part of the language, no separate type is necessary |
| Boolean | `boolean` |
| String | `String` |

The type *names*, as appearing in the PREMO document, are not reused literally, and this may be a source of confusion for whoever wants to consult the original PREMO document while reading this book. The main reason is the relative inflexibility of Java in this respect. Indeed, whereas in C or C++ it would be possible to say, e.g.,

```
typedef long Time;
```

which just gives a new name to an existing type, this facility does not exist in Java. The only other possibility would have been to wrap all PREMO non–object types into separate Java classes for the purpose of renaming, which would have been an unnecessary complication. In this book, we refer to the Java names only.

The "Time" non–object type is used to describe the ticks of a clock, and plays an important role in multimedia synchronization. PREMO gives the implementors the choice of choosing either a real number or a large integer number to represent time. We have chosen the latter, based on the fact that Java's view on elapsed time (see, e.g., the `currentTimeMillis` method of the `java.lang.System` object) uses long integers, too.

The PREMO standard makes extensive use of a non–object data type called *Value*, which acts as a union type encompassing all of the other PREMO non–object data types, including object references. Java does not provide a union constructor over its basic data types. Instead, our PREMO Java binding uses the Java class `Object` in place of value. PREMO non–object values (which are basic values in Java) are then represented by instances of the so–called Java "Envelope" classes when they appear as values of this union type. For example, a value of type `int`, when used in the union type, is represented as an object of type `java.lang.Integer`.

### 5.2.2 Constructed Data Types

Creating arrays is probably the most important non–object data mechanism used in PREMO. These are represented by Java arrays. Internally, when variable length arrays are necessary, `java.util.Vector` is also used, but this does not appear on the level of interface specifications.

Another mechanism is the creation of simple classes, called structures, for collecting attributes (i.e., public variables) of other non–object types. The only method implemented in a structure is the `equals` method (automatically inherited from `java.lang.Object`) to ensure a proper use (comparing the constituent attributes rather than the object references), as well as obvious constructors to fill the attribute values. They may also appear as nested top–level classes as in the following example:

```
public class Structure extends SimplePREMOObject {
   public static class SomeData {
      public String key;
      public Object value;
   }
   public SomeData[] someData;
}
```

(the example is fictitious). Using the nested `SomeData` class in the example makes the specification cleaner and simpler. The data itself can be accessed through a statement such as:

```
obj.someData[0].key
```

where ‘`obj`’ is an object of type `Structure`.

A further construction mechanism, widely used in the PREMO specification, is enumeration[1]. Formally, this means defining a type whose value is restricted to a finite (usually small) set of symbols. For example, PREMO defines the enumeration

$$ActionType ::= Enter \mid Leave$$

to denote a type which may have only two values, symbolically denoted by “*Enter*” and “*Leave*”, respectively. The usual counterpart for this data type in programming languages is what is denoted as `enum` in C or C++. Because this construction does not exist in Java, a separate mechanism had to be constructed to implement PREMO enumerations with Java classes. The public interface of the common superclass, which is defined in `premo.impl.utils`, is as follows:

```
package premo.impl.utils;
public abstract class PREMOEnumeration implements java.io.Serializable
{
   protected PREMOEnumeration();
   public boolean equals(Object obj);
}
```

Note that the constructor of the object is protected, i.e., clients, in general, cannot construct instances of this class directly. Using this superclass, a PREMO enumeration is defined as:

```
package premo.std.part2;
public final class ActionType
   extends premo.impl.utils.PREMOEnumeration {
   public static ActionType Enter;
   public static ActionType Leave;
   private ActionType(int i) { super(i); }
   static {
      Enter = new ActionType(0);
      Leave = new ActionType(1);
   }
}
```

[1] Not to be confused with the `java.util.Enumeration` interface!

The `static` section of the class creates a fixed number of enumeration instances. The names of these instances constitute the enumeration tags. The separate integer value is used to differentiate among the various enumeration instances, and is used internally in the `equals` operation (inherited from `PREMOEnumeration`). Because the constructor is private, clients do not have the ability to define new instances, which corresponds to the fact that enumerations may only have a finite, pre–defined set of values. In the rest of the book, when defining a PREMO enumeration, only the constants will be listed, and we will omit the standard constructor and the `static` section.

Part 2 of PREMO defines a number of constructed non–object types (as all other PREMO Parts do). To make the text more readable, these specifications will be presented together with the objects which use them.

### 5.2.3 Exceptions

Exceptions in PREMO are simple extensions of the standard Java exceptions. A common superclass for all PREMO exceptions is defined as follows:

```
package premo.impl.utils;
public abstract class PREMOException
   extends java.lang.RuntimeException implements java.io.Serializable
{
   public Object[] Val;
   public PREMOException();
   public PREMOException(String s);
}
```

which is used by all exceptions defined by PREMO. This class also has an extra attribute `Val`. This is used by some PREMO objects to add additional information to the exception they throw. Another noteworthy aspect of this class is that it extends `java.lang.RuntimeException`, rather than `java.lang.Exception`. This means that methods need not declare these exception in their `throws` clause (although we will always do it to make the specifications more complete) and, more importantly, the invocation of operations raising these exceptions are not obliged to be enclosed in a `try–catch` pair.

The PREMO exceptions themselves are simple subtypes of `PREMOException`, defined in the same way as the exceptions in the standard Java packages. Part 2 defines the following exceptions:

```
CannotMeetCapabilities          NoKey
IncorrectInit                   NotInTypeGraph
InvalidCapabilities             OperationNotDefined
InvalidKey                      ReadOnlyProperty
InvalidReference                RepeatedEvent
InvalidType                     WrongState
InvalidValue                    WrongValue
```

The semantics assigned to throwing these exceptions are explained in conjunction with the object methods throwing them.

## 5.3 Top Layer of the PREMO Object Hierarchy

### 5.3.1 The `PREMOObject` Interface

All objects in the PREMO object hierarchy implement the `PREMOObject` interface. In other words, this type represents the “root” of the entire PREMO hierarchy.

`PREMOObject` defines three methods which, by virtue of inheritance and the implementation of interfaces, are available for all PREMO objects. Here is the complete specification of `PREMOObject`:

```
package premo.std.part2;
import java.lang.Class;
public interface PREMOObject extends java.rmi.Remote {
   Class    inquireType();
   Class[]  inquireTypeGraph();
   Class[]  inquireImmediateSupertypes();
}
```

The methods return the class of the object, the sequence of all supertypes, and the sequence of immediate supertypes, respectively. These methods are very similar to analogous to methods describing the class hierarchy of Java classes, and available in the standard `java.lang.Class` object. The ones defined in `PREMOObject` are complimentary in the sense that all returned information refers to types *specified in the* PREMO *document only*, i.e., the various implementation dependent classes and interfaces are filtered out. Of course, if knowledge of the complete inheritance and implementation hierarchy is necessary for the client, the `java.lang.Class` methods are always available.

Although not appearing in the `PREMOObject` interface (a Java interface specification cannot contain protected methods), PREMO also defines three protected operations which are to be implemented by the classes implementing the `PREMOObject` interface. These are:

```
protected void initialize(Object initValue) throws IncorrectInit;
protected void initializeOnCopy()
protected void destruct()
```

These methods are invoked when the object is created, cloned, or destroyed, respectively. In terms of Java, they are essentially constructors (with a fixed signature), clone operation, and finalizers, albeit defined in a language independent way.

The complete PREMO hierarchy branches off from `PREMOObject` into three large categories of PREMO objects: so–called simple PREMO objects, callbacks, and enhanced PREMO objects (see Figure 5-1). Simple PREMO objects are essentially data structures which, in contrast to non–object data types, can be folded into a PREMO subtyping hierarchy. Callbacks consist of two interfaces only, and represents therefore a fairly small category. The real multimedia service objects form the category of the enhanced PREMO objects and, not surprisingly, the bulk of the PREMO document is devoted to the specification of various enhanced PREMO objects. In what follows, the type structure of all three categories will be presented.

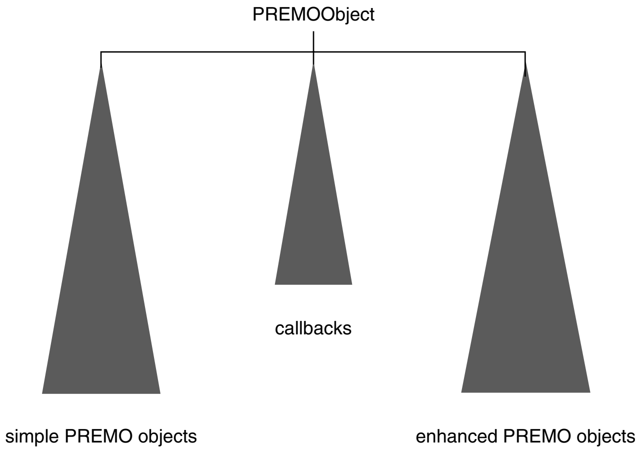


Figure 5-1 — Main Categories of PREMO Objects

### 5.3.2 Simple PREMO Objects

Simple PREMO objects are, conceptually, data structures needed for the proper specification of various multimedia service objects. They are very similar to constructed non–object data type classes (see section 5.2.2) in the sense that the intention is a compact representation of entities such as a geometric point, an event, or a constraint specification. The similarity is also reinforced by the fact that simple PREMO objects do not define multimedia *services*, e.g., over a network. The major difference between constructed data types and simple PREMO objects is that the latter are part of the full PREMO hierarchy, and they can also be subject to various subtyping patterns. For example, it is possible to build a complete hierarchy of various events using subtyping (this is done very frequently in various interactive systems).[1)]

Formally, simple PREMO objects are defined to be subclasses of the following class:

```
package premo.std.part2;
public abstract class SimplePREMOObject
   implements PREMOObject, java.io.Serializable {
}
```

The class is abstract, i.e., non–instantiable, and it is a direct subtype, in the PREMO sense, of `PREMOObject`. Note that the class also implements the standard `java.io.Serializable` interface, i.e., instances of this class can appear, for example, as arguments of remote object calls.

[1)] It must be noted that the strong similarity between simple PREMO objects and constructed data types is an artefact of the nature of Java, and not of the PREMO specification proper. Other languages may offer much richer data structuring possibilities, such as enumerations and data structures. In this case, most of the PREMO non–object data types could be described independently of the object/class hierarchy.

The various simple PREMO objects are all subclasses from `SimplePREMOObject`, either directly or indirectly. To reinforce the "data" nature of these classes, they are usually defined in terms of public variables and not methods. These variables are also referred to as "structure tags", and simple PREMO objects are also referred to as "structures".[1)]

Part 2 of PREMO defines only four simple PREMO objects. Other parts of PREMO, especially Part 4, make a much more elaborate use of them. Two of these simple objects, `ActionElement` and `SyncElement`, are closely related to the semantics of other objects, such as the `Controller` in the case of `ActionElement`, and the synchronization objects in the case of `SyncElement`. They will be defined in later sections. The two other simple PREMO objects, defined in Part 2, are of a very broad use in the standard, so it is better to define them in general. They are also very good examples of how the various constructions described so far converge in concrete type specifications.

#### 5.3.2.1 Event Structures

Event handling and event management play a central role in all dynamic systems, including PREMO. Events represent basic building blocks to convey information through the system in an asynchronous fashion. Events are used to manage interaction, synchronization patterns, to monitor various activities in other objects, etc.

The complete event handling model in PREMO involves several different objects, which will be defined later (see section 5.4.1). All these objects make use of the simple PREMO object `Event`, which carries the necessary information. An `Event` structure has a *name*, which can be used to identify the event itself, it contains an *event data* which, at the abstract level, consists of an array of key–value pairs, and an *event source*, which is a reference to the object which has created ("raised") the event instance.

Here is the class specification of `Event`:

```
package premo.std.part2;
import java.lang.*;
public class Event extends SimplePREMOObject {
   public String eventName;
   public static class EventData implements java.io.Serializable {
      public String key;
      public Object value;
   }
   public EventData[]         eventData;
   public EnhancedPREMOObject eventSource;
}
```

Note the use of a nested top–level class for the specification of a simple data structure (see also page 61). The type `EnhancedPREMOObject` is the "root" of all enhanced PREMO objects, see section 5.3.4.

[1)] Of course, an implementation may also redefine operations such as `equals`, inherited from `java.lang.Object`.

#### 5.3.2.2 Constraint Structures

The term “constraint” is used in a restricted sense in PREMO, and does not refer to some kind of a complex constraint management system. It rather refers to a conceptually simple, albeit extremely important set of operations over values of different types, leading to boolean results. Just such as events, these constraints appear at various places in PREMO, e.g., in controlling the creation of PREMO objects, in managing and inquiring their properties, etc.

A constraint structure contains a key–value pair, which must be compared to some other key–value pairs, and the description of the comparison operator itself. The latter is simply an enumeration of operations such as equal, not equal, greater than, includes, excludes, etc. which are to be applied to the values of the keys.

Formally, the `Constraint` structure makes use of the following enumeration:

```
package premo.std.part2;
public final class ConstraintOp
   extends premo.impl.utils.PREMOEnumeration {
   public static ConstraintOp Equal;
   public static ConstraintOp NotEqual;
   public static ConstraintOp GreaterThan;
   public static ConstraintOp GreaterThanOrEqual;
   public static ConstraintOp LessThan;
   public static ConstraintOp LessThanOrEqual;
   public static ConstraintOp Prefix;
   public static ConstraintOp Suffix;
   public static ConstraintOp NotPrefix;
   public static ConstraintOp NotSuffix;
   public static ConstraintOp Includes;
   public static ConstraintOp Excludes;
}
```

(see page 62 for the description of `PREMOEnumeration`). Using this enumeration, the specification of a `Constraint` structure is as follows:

```
package premo.std.part2;
import java.lang.*;
public class Constraint extends SimplePREMOObject {
   public ConstraintOp constraintOp;
   public static class KeyValue implements java.io.Serializable {
      public String key;
      public Object value;
   }
   public KeyValue keyValue;
}
```

(Note that arrays can also be represented as `Object` in Java, so this structure may also store a full sequence of values associated with a key.)

### 5.3.3 Callbacks

Management of events dynamically is usually achieved by "registering interest" in some events. This is done by publishing the reference of the interested party. The object which raises or forwards the event may then notify the interested party of the occurrence of the event.

The `Callback` interface is defined to facilitate this mechanism, by defining a single entry point for any interested party. The interface has the following, very simple definition:

```
package premo.std.part2;
public interface Callback extends PREMOObject {
   void callback(Event callbackValue);
}
```

Various enhanced PREMO objects implement this interface, thereby assigning a specific behaviour to the `callback` operation.

Although the interface specification is simple, there is an underlying semantics to the operation `callback` which must be taken into account when the interface is implemented. Two features should be emphasized:

- If the operation's semantics is such that the callback value is forwarded to a third party, or the structure tags are changed, a copy of the event structure should be made. Indeed, the same event structure can be forwarded (through the `callback` operation) to a number of objects, whose identity and number is not known in advance. Changing the callback values may lead to uncontrollable situations.
- Callbacks are usually used for interaction and synchronization. In other words, in time critical situations. It is therefore of a paramount importance that the caller to the `callback` operation *is not suspended* for a long time while the operation performs its own activity. The `callback` operation is therefore defined to be asynchronous (see Section 3.8).

Whereas, in simple cases, the semantics of the `callback` operation may be defined to affect the state of the object directly, it is very often the case that this operation acts only as an entry point to call other operations on the object. To facilitate this second case, PREMO also defines an interface which extends `Callback`, called `CallbackByName`. This extension does not add any new methods, but overrides the (inherited) asynchronous `callback` operation:

```
package premo.std.part2;
public interface CallbackByName extends Callback {
   void callback(Event callbackValue) throws OperationNotDefined;
}
```

The `callback` of `CallbackByName` has the following behaviour: the `eventName` structure tag of the `Event` structure (appearing as the input argument of `callback`) is interpreted to be the name of a local operation which is then internally invoked by the `callback` operation (an exception is raised if the name does not refer to any valid operation). By default, all other structure tags of the `Event` structure are disregarded by the `callback` operation. Subtypes of `CallbackByName` may add an additional behaviour to the operation which also takes these tags into consideration.

### 5.3.4 Enhanced PREMO Objects

Most of the objects defined by PREMO are enhanced PREMO objects. All other categories of objects, as well as the various non–object types, are defined and used in order to make the specification of enhanced PREMO objects concise and precise. Enhanced PREMO objects have a common supertype within the PREMO hierarchy called, not surprisingly, `EnhancedPREMOObject`.

#### 5.3.4.1 Enhanced PREMO Objects as Service Objects

A fundamental restriction of PREMO is that *enhanced PREMO objects, and only those, offer services* over a distributed PREMO environment. In terms of our implementation strategy, based on Java RMI, this means that enhanced PREMO objects, and only those, should be registered as RMI server objects. This means that the “dual” implementation structure, as described in section 4.2.1, is valid for all enhanced PREMO objects.

Formally, enhanced PREMO objects are those which implement an interface, called `EnhancedPREMOObject`, defined as follows:

```
package premo.std.part2;
import java.lang.*;
public interface EnhancedPREMOObject
   extends PREMOObject, java.rmi.Remote {
}
```

[Note: we have omitted the various methods defined by `EnhancedPREMOObject` for now, see section 5.3.4.2]. Note that the interface also extends `java.rmi.Remote`, which is necessary to ensure that the object could serve as an RMI server object.

This interface is implemented by a separate class in the `premo.impl.part2` package:

```
package premo.impl.part2;
import premo.std.part2.*;
public abstract class EnhancedPREMOObject_Impl
   implements EnhancedPREMOObject {
}
```

The various enhanced PREMO objects are implemented by classes extending, either directly or indirectly, this class (and implementing their corresponding interface, of course).

#### 5.3.4.2 Property Management

Properties are used to store values with an object that may be dynamically defined and are outside of the type system. Properties are pairs of keys (i.e., strings) and arrays of values which are conceptually stored within an enhanced PREMO object (to use another terminology, each enhanced PREMO object has an associated dictionary). Operations are introduced to define, delete, and inquire values from the array associated with a key. Properties can be used to implement various naming mechanisms, store information on the location of the object in a network, create annotations on object instances, and they also play an essential role in negotiation mechanisms within PREMO (section 5.6). The existence of some properties (i.e., the keys) may be stipulated by the standard for a specific object type, but clients can attach new properties to objects at any time.

Properties may be defined as read only. This means that they cannot be defined through an operation on the object, nor can they, or their associated values, be changed or deleted. Read only properties are typically set by the object when initialized, and are used to describe the various capabilities of the object.

Why use properties? The fundamental reason lies, in fact, in the conservative nature of the PREMO object model. Indeed, in PREMO, operations on a type are defined statically, when defining ("declaring") the object. Once the object type has been defined, and an object instance of that type is created, no new operation can be added to that object instance dynamically. On the other hand, it has been advocated elsewhere that more dynamic object models should be used for graphics or multimedia (see, e.g., [12] or [42]). Indeed, the use of delegation[62] or, on a more "modest" level, a more dynamic view of objects such as, for example, the approach adopted in Python[87] (which allows the addition of operations dynamically), would be more appropriate for graphics and multimedia systems. These features would play an important role, for example, in constraint management, in the adaptability of objects, etc. However, experience has also shown that implementing such features on top of languages or environments which are not prepared for them represents a significant burden and leads to a loss of efficiency. And, unfortunately, none of the widespread object–oriented systems or languages (C++, OMG specifications, Java, etc.) implement delegation or anything similar. As a consequence, and after some discussion, the adoption of such features was rejected for the development of PREMO.

Properties aim at offering a replacement for such advanced features on a lower level. Although properties do not allow new operations to be added to an object instance, the mechanism can at least be used to simulate adding and manipulating new attributes (essentially, data) to object instances. Obviously, the implementation of properties does not represent a significant problem. The dynamic nature of properties is quite beneficial to PREMO. This will become clear in later chapters. One could therefore say that properties play a somewhat less elegant, but very useful role in PREMO in increasing the dynamic nature of object instances.

Basic property management can be carried out with a set of methods defined in the `EnhancedPREMOObject` interface, thereby available for all enhanced PREMO objects, and implemented in the `EnhancedPREMOObject_Impl` class. In the remainder of this section, we will go through these operations in somewhat more details.

#### *5.3.4.2.1 Property Definition*

The `EnhancedPREMOObject` interface contains the following operations to create or to modify properties.

```
void defineProperty(String key, Object[] value)
   throws ReadOnlyProperty;
```

This method adds a new property to the object. If the key identifies a property already defined for the object, the new value is assigned to the property, replacing the previous value(s). Otherwise, a new property is created with key and value.

```
void addValue(String key, Object value)
   throws ReadOnlyProperty;
```

This method adds a value to the properties for the argument key. If the key has not been used yet, a new property is defined. Both of these methods may raise an exception if the key refers to a read–only property.

By default, if a property value is defined for a key which already exists for the object instance, the old value is silently overridden. However, the client has the ability to add a reference to a `Callback` object to a property key to monitor those changes. The callback is activated whenever a new value is defined for the key. Adding a callback reference is done through the method:

```
void setPropertyCallback( String   key,
                          Callback callback,
                          String   eventName )
   throws NoKey;
```

The newly created event instance, forwarded to the callback, uses the name given in the method argument. The event structure contains the key–value pair corresponding to the new setting.

#### *5.3.4.2.2 Removal of Properties*

Two methods are defined to remove properties from an object. The method

```
void undefineProperty(String key)
   throws ReadOnlyProperty, NoKey;
```

removes the property altogether, deleting both the key and all the corresponding values, whereas the operation

```
void removeValue(String key, Object value)
   throws ReadOnlyProperty, NoKey, InvalidValue;
```

removes a single value from the property defined for a key. (The exception `Invalid-Value` is raised if the value does not appear on the property list.) Both of these operations raise an exception if a read only property is referred to in their argument.

#### *5.3.4.2.3 Property Inquiry Operations*

A single property can be inquired through the

```
Object[] getProperty(String key) throws NoKey;
```

method. If all the properties are to be inquired, they can also be accessed through the

```
PropertyPair[] getPairs();
```

method, where `PropertyPair` is a separately defined non–object data type of the form:

```
public final class PropertyPair implements Serializable {
   public String    key;
   public Object[]  value;
}
```

Most of the methods so far could raise an exception if the key was not present, or it referred to a read–only property. The full set of keys can also be retrieved through the

```
public static class KeyInfo {
   public String  key;
   public boolean readOnly;
}
KeyInfo[] inquireProperties();
```

method, which may aid the client to form a proper call sequence.

### *5.3.4.2.4 Property Matching*

Property matching is the most powerful operation among the property management methods of the `EnhancedPREMOObject` interface. It allows for a constraint–based retrieval of properties, serving as a basis for various negotiation mechanisms occurring in multimedia systems.

The interface specification of the method is as follows:

```
public static class MatchPropertyResults {
   public PropertyPair[]   satisfied;
   public PropertyPair[]   unsatisfied;
}
MatchPropertyResults matchProperties(Constraint[] constraintList);
```

(the `Constraint` structure is defined in Chapter 5.3.2.2 on page 67, and `MatchPropertyResults` is a top–level nested class.) Semantically, what happens is as follows: The properties defined for the object are matched against the property sequences in `constraintList`. For each key appearing in this constraint list, the values are compared against the value or values stored with an identical key in the object. Comparison is based on the boolean operation defined by the enumeration `ConstraintOp` (also defined in Chapter 5.3.2.2), and appearing as the structure tag of constraint list. The left operand of the operation is the property stored in the object, and the right operand of the operation is the value appearing in the `constraintList` structure. If the operation does not make sense, the result of the comparison is `false` (for example, "`Includes`" for numerical types).

The `satisfied` array contains those keys with associated values for which the comparison has resulted in `true`. The `unsatisfied` array contains those keys with associated values for which the comparison has resulted in `false`.

An example will clarify the use of this method. A fictitious audio object may store the various audio formats it can decode in a property list. The object providing the service may define a (read–only) sequence of values for the key "`AudioFormatK`", e.g., <"`AIFF`", "`AIFC`">, describing the audio file formats it can recognise. The `matchProperties` method may be invoked with a pair consisting of a key and a value:

```
["AudioFormatK", "AIFF"]
```

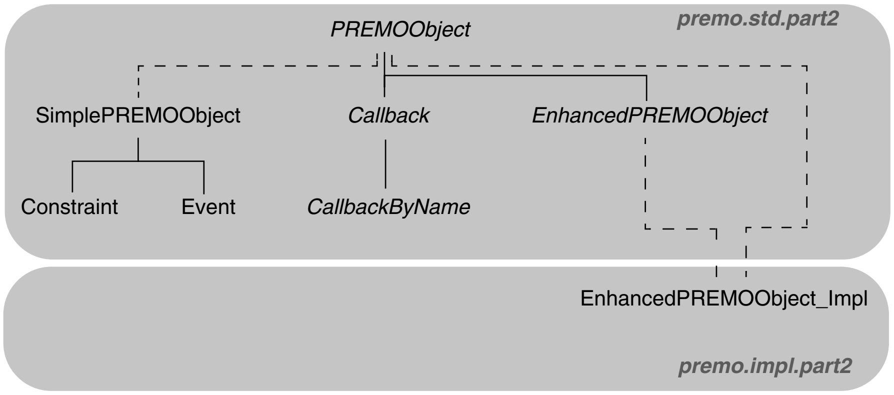


Figure 5-2 — Top Classes and Interfaces of PREMO

using the comparison operator “`Equal`”. The result will be:

```
satisfied: ["AudioFormatK", <"AIFF">]
unsatisfied: ["AudioFormatK", <"AIFC">]
```

Another call, using:

```
["AudioFormatK", "IRCAM"]
```

will result in

```
satisfied: ["AudioFormatK", <> ]
unsatisfied: ["AudioFormatK", <"AIFF","AIFC">]
```

etc. Based on this information the client can choose the `AIFF` file format which can be managed both by itself and the audio service.

The example can be made more complex. For example, using more than one key in the invocation of the `matchProperties` operation (e.g., also include sampling size), and optimizing the calls through the use of arrays of values and the “`Includes`” operator instead of “`Equal`”, further information can be retrieved on the object. This can serve as a basis for powerful negotiations.

### 5.3.5 Top Layer of PREMO

Figure 5-2 gives an overview of all the interface and class definitions appearing at the top level of the PREMO type hierarchy and is a detailed version of Figure 5-1. Only those simple objects are depicted on the figure which have already been presented.

## 5.4 General Utility Objects

We have, somewhat arbitrarily, re–grouped a few PREMO Part 2 objects under a category called general utility objects, although this categorization does not appear in the original Standard itself. The objects in this category provide some elementary building blocks which are used in various other places in PREMO, but they are rarely used in isolation, i.e., without being bound to some other, more complex objects. There are three groups of general utility objects:

- Event handler objects, which provide an event propagation mechanism in PREMO.
- Controller objects, providing an interface for controlled finite state machines.
- Timer objects, which define the interfaces to measure time in PREMO.

This section will present these objects in more detail.

### 5.4.1 Event Management

Forwarding information, i.e., data, through operation invocation is a relatively static action. The caller has a direct knowledge of the callee (modulo the actual value of an object reference), only one callee can be invoked at a time, the callee has no real control over the occurrence of the call, etc. Whereas this approach to information transfer is appropriate most of the time, it has long been recognized that dynamic systems need a more flexible way of forwarding information, too. As opposed to a direct call, this dynamic form of data transfer should be such that:

- The caller, or the source of the information, should be separated from the callee, or the possible consumer of the data. The caller does not need to know about the consumer of the data.
- Data transfer should be as asynchronous as possible. The source of the information should just make the data “known” to its environment, and continue its own activities.
- It should be possible to have more than one consumer at a time.
- The receiver should have the ability to dynamically control whether it is interested in the information or not.

*Event handling*, or *event management*, has become the standard answer to these concerns, and event management is ubiquitous in dynamic and interactive systems nowadays. Events were already present as early as 1982, when the first version of GKS became more widely available as a technical document. The GKS standard included the notion of event mode input which had some of the characteristics described above[5]. The notion of events became more familiar through various windowing environments, such as X11, and is now part of almost all graphics and multimedia systems. Java’s AWT has the notion of events, too, and the listener–based event model, present in AWT since Java version 1.1, shares a lot of common characteristics with what will be described below for PREMO.

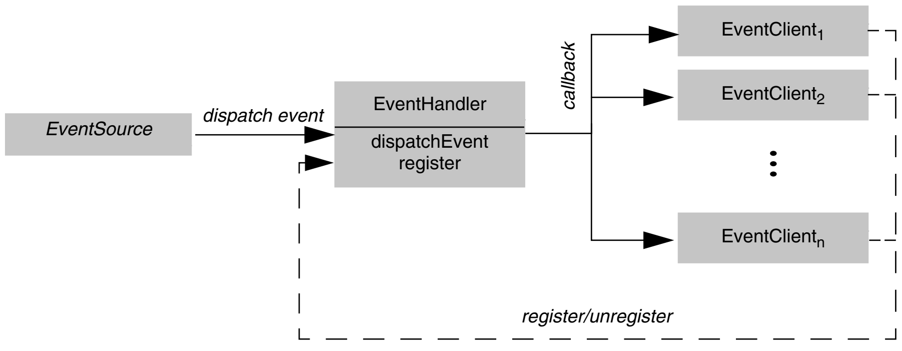


Figure 5-3 — Event Management in PREMO

Just as in the case of a precise object model, various systems have their own view of event management, but none of the present schemes suffices for PREMO. Consequently, PREMO defines its own event management model.

### 5.4.1.1 The PREMO Event Model

Figure 5-3 gives a rough overview of the main notions involved. *Events* are simple PREMO objects which were already defined in section 5.3.2.1 on page 66. Events have a *name*, a *source*, and they may contain *event data.* A *name* is also referred to as event *type*, which is a `String`. A *source* is the reference to a PREMO object (usually a reference to the object which creates a specific instance). The *event data* is conatined in a sequence of key–value pairs, much like properties. Event management is concerned about how these units of information are propagated among PREMO objects.

The main feature of event management is the separation between the source of the events and the objects receiving the events, called also *event clients*. This separation is done through the use of a special object in PREMO, called an *event handler.* Objects, which intend to propagate an event, send the event instance to this event handler object. and it is the event handler's task to broadcast the event instance to a set of event clients. Objects, which want to become event clients, *register* to the event handler. If they do not want to be an event client any more, they can *unregister.* In other words, event handlers embody a one–to–many propagation of events with a dynamic management of prospective event receivers. The event source does not know who the event recipients are for a specific event; it only sends the event to the event handler.

Prospective event clients can register themselves to several event handler instances, thereby having some control over which category of events they want to receive. However, registration, and the corresponding event propagation, is much more finely tuned. Indeed:

1. When registering, the client gives an event name to the event handler, too. When a new event arrives, the event handler compares the name of the event to the names which are part of client registrations. Only those registrations are considered where

these names coincide. In other words, the client registers its interest for *a specific type of event*. If the client is interested in several different event types from the same event handler, it should register separately for each of them.

2. Beyond the event name, clients can also impose some constraints on the event data as part of registration. These constraints are checked by the event handler by comparing the event data to the constraints. If the result of this comparison is `false`, the event is not propagated to that event client. As a simple example, the client may specify that it is interested in an event of a specific type if and only if its data contains an entry with a specific key and value. More complicated constraints are also possible.

Prospective event client objects should implement the `Callback` interface (see section 5.3.3). Event propagation is done by the event handler through the `callback` operation. The event handler object is defined to be a `Callback` object, too, with the `callback` object identified with the event dispatch operation. This means that various event handler objects may be "chained", thereby forming a complex network of event propagation patterns.

Event handlers are frequently used as building blocks for interaction. The requirement of asynchronicity in event propagation is therefore essential. The main action of the event handler, i.e., managing the constraints and effectively propagate the events, should never suspend the event source which intends to dispatch a new event. The activity of the event handler is therefore of a primary importance: conceptually, the real effect of the `dispatchEvent` is simply to place the event instance into some sort of internal event queue and a separate thread should be responsible for emptying this queue and propagate the events. In other words, this operation is defined to by *asynchronous*.

Note that the PREMO event model is not unlike the event model used by Java in AWT or in the Java Beans specification. The major difference is that, although the "philosophy" is similar, the mechanism is more explicit in PREMO than in Java. (Indeed, in the latter, the various AWT components play the role of both the event sources and the event handlers, rather than separating the two into different object types.)

#### 5.4.1.2 The Event Handler Object

The "core" of the PREMO event management mechanism is the `EventHandler` object type. The interface of the object is as follows:

```
public interface EventHandler
    extends Callback, EnhancedPREMOObject, java.rmi.Remote
{
   long register( String eventType, Constraint[] constraints,
                  AndOr matchMode, Callback theCallback);
   void unregister(long id) throws InvalidEventId;
   void dispatchEvent(Event e);
}
```

The `dispatchEvent` operation is defined to be asynchronous. Event registration makes use of the `Constraint` structure, defined in section 5.3.2.2 (see page 67) and a simple PREMO enumeration:

```
public final class AndOr
   extends premo.impl.utils.PREMOEnumeration {
      public static AndOr And;
      public static AndOr Or;
}
```

The registration operation returns a registration identifier. This identifier should be used to unregister (an exception is raised if an invalid event registration identifier is used as argument for unregistration).

Dispatching an event is achieved through the `dispatchEvent` call although, as previously stated, the "real" effect of this operation is merely to put a copy of the event structure into an internal queue; actual dispatch is done in a separate thread. The implementation of the `EventHandler` interface must provide an implementation for the `callback` operation, too (in order to implement the `Callback` interface). The effect of `callback` is identical to `dispatchEvent` in this case. The two operations can be considered as simple synonyms.

The details of event propagation are at the heart of the object's semantics. Here is what the object has to do when a new event instance is received:

1. The type of the new event is compared to all registrations. Only those are retained where the new event's type matches with the registered event type.
2. For all retained registrations:
   2.1. If either the event data or the registered constraint array is empty, the event is forwarded. This is done by invoking the `callback` operation on the registered object with the event instance as an argument.
   2.2. If the arrays are not empty, all key–value pairs with identical keys, in the constraint and the event data arrays, respectively, are compared. Comparison means using the operation defined in the `ConstraintOp` field of the constraint (which tells whether the comparison means 'equal', 'greater than', 'contains', etc., see page 67). The left operand of the comparison operator is the value stored in the event handler, i.e., the registered value, and the right operand is the value in the new event instance.[1)]
   2.3. If the registered value for the `AndOr` enumeration is `And`, the logical `AND` of all comparisons is considered. Otherwise, the `OR` is considered. This leads to the result of the full constraint checking: If `true`, the event is forwarded (like in 2.1 above). If `false`, this registration is not considered for event dispatch for this event instance.

Let us see some examples. In a very simple case, registration with no constraints is used:

```
long id = eHandler.register("PREMOEvent",null,null,this);
```

[1)] If the operation does not make sense, e.g., a 'contains' is required for two numerical types, the result is `false`.

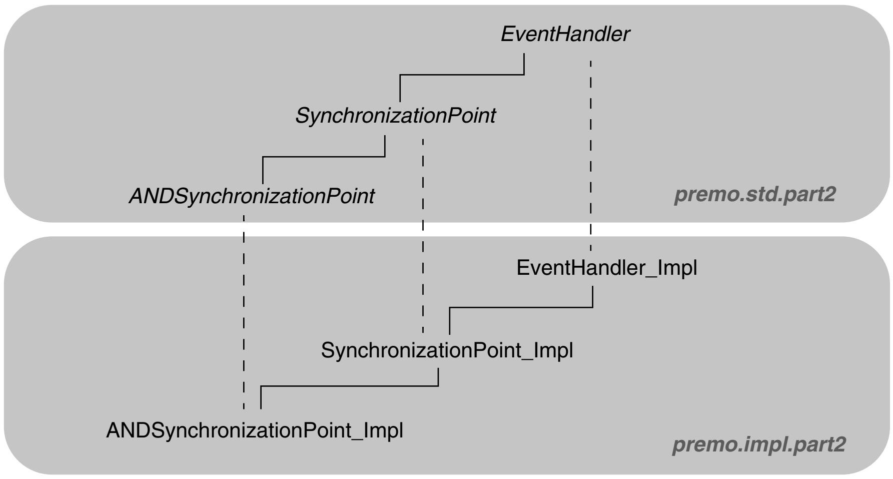


Figure 5-4 — Event Handler Objects

which is issued by the prospective event client on an event handler. The event source object might perform the following set of operations:

```
Event ev       = new Event();
ev.eventName   = "PREMOEvent";
ev.eventSource = this;
ev.eventData   = new Event.EventData[] { new Event.EventData() };
ev.eventData[0].key     = "MyKey";
ev.eventData[0].value   = "MyValue";
eHandler.dispatchEvent(ev);
```

which creates an event instance with an event data array of length 1. The new event instance is dispatched. Because the event client has not defined any constraints, it will receive a copy of the event `ev`. If, however, event registration were done through the following sequence:

```
Constraint[] cons    = new Constraint[] { new Constraint() };
cons[0].key          = "MyKey";
cons[0].value        = "AnotherValue";
cons[0].constraintOp = new ConstraintOp(ConstraintOp.Equal);
long id = eHandler.register("PREMOEvent",cons,AndOr.And,this);
```

Then the client will *not* receive a copy of `ev`. Indeed, the constraint imposed on the key "`MyKey`" is not fulfilled in this case.

### 5.4.1.3 Synchronization Points

General event handler objects, as described in the previous paragraph, do not impose any restrictions on the type of events which they are ready to forward. Any event is accepted and forwarded, subject of course to the constraints imposed on registration.

Synchronization points are specialized event handlers which do impose some further restrictions on their "dispatch" side. A synchronization point object maintains an internal table of events, referred to as "synchronization events". Operations are defined to add and delete a synchronization event. These events are those which are accepted by the synchronization point for event propagation. Dispatching an event which is not a synchronization event results in an exception and the event is *not* forwarded. (One must be somewhat more precise about what it means that an event is a synchronization event or not. The internal table of synchronization events contains a set of object references; an event, which appears as an argument to the `dispatchEvent` operation, must be *equal* to one of the events in the table. Equality means that the event source refers to the same object, the event names are identical, and that the event data are equal[1].)

Note the fact that event *sources* should be equal for dispatch (a reference to the event source is part of an event structure): this means that synchronization objects accept events from specific objects only. Hence their name — these objects are used in multimedia synchronization schemes where they synchronize information flow among specific object instances.

The interface of a synchronization point object is quite straightforward:

```
public interface SynchronizationPoint
   extends EventHandler, java.rmi.Remote
{
   void addSyncEvent(Event e)    throws RepeatedEvent;
   void deleteSyncEvent(Event e) throws UnknownEvent;
   long register( String eventType, Constraint[] constraints,
                  AndOr matchMode, Callback theCallback)
      throws InvalidEventId;
   void dispatchEvent(Event e)   throws UnknownEvent;
}
```

The operations `addSyncEvent` and `deleteSyncEvent` are used to add, respectively delete, a synchronization event. An exception is raised by `addSyncEvent` if an event has already been added as a synchronization event.

The operations `register` and `dispatchEvent` are inherited from `EventHandler`, but their semantics is extended slightly, and they may throw exceptions which their ancestors don't. In the case of a synchronization point, the event type must coincide with the event type of at least one of the synchronization events (an event of another type would not be forwarded anyway), and this is checked by the `register` operation before performing the "original" registration. The `dispatchEvent` operation (which overloads the operation inherited from `EventHandler`) checks its argument and raises an `UnknownException` if the event is not a synchronization event, further disregarding the event. Otherwise, the original (inherited) meaning of the operation comes into effect.[2]

[1] Eventually, what this boils down to, is the equality of the values in the event data, i.e., the equality of the objects the values represent. In the case of Java, one has to define the operation `equals` to give a more specific meaning to equality.

[2] Note a very subtle difference between the operations `callback` and `dispatchEvent`. The callback is not listed here, i.e., the extra exception is not raised (although a non–synchronization event is disregarded all the same). This is because, when using callback, the external world would consider a general `Callback` object only, and it cannot be expected to check this special exception.

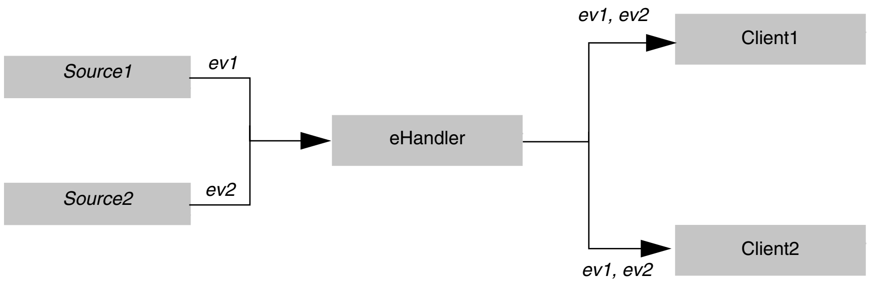


Figure 5-5 — Synchronization Points

A further specialization of the synchronization point is the AND synchronization point. This object does not add any new operation to the synchronization objects, i.e., its specification is simply:

```
public interface ANDSynchronizationPoint
   extends SynchronizationPoint, java.rmi.Remote {
}
```

However, while implementing the semantics of synchronization points, this object also *delays* event propagation. The goal is to "collect" a number of incoming events, which are identical *except* that they originate from different event sources. Only when all such events arrive to the AND synchronization point would the object forward the accumulated events. This is done by, conceptually, categorizing all synchronization events by event name and event data. All synchronization events, having identical name and data, fall into the same category, *regardless* of their source. When an event arrives for dispatching, its category is located and, within its category, the synchronization event itself found. However, instead of being dispatched, this information is just flagged. When *all* events within a category are flagged, *and only then*, will all the events be dispatched (following, of course, the original semantics of dispatching, i.e., the constraint mechanism may still block the propagation of a specific event instance for a specific client).

The `ANDSynchronizationPoint` can be functionally described as an array of AND components, each of which requires all registered events for a given category to be collected before dispatching these events to registered clients. It is important to note that multiple categories of events can be handled within one `ANDSynchronizationPoint` instance.

As an example, let us consider the following code (see also Figure 5-5). First of all, two event registrations are performed, both with the event names "`PREMOEvent`", one for the client `Client1` and the other for `Client2`. To simplify the code, no constraints are added to the registration:

```
long id1 = eHandler.register("PREMOEvent", null, null, Client1);
long id2 = eHandler.register("PREMOEvent", null, null, Client2);
```

Then, two event instances, both having "`PREMOEvent`" as a type, are created, for the sake of the example, with empty data. Also, both events are registered by the synchronization point as synchronization events:

```
Event ev1          = new Event();
ev1.eventName      = "PREMOEvent";
ev1.eventSource    = Source1;
Event ev2          = new Event();
ev2.eventName      = "PREMOEvent";
ev2.eventSource    = Source2;
eHandler.addSyncEvent(ev1);
eHandler.addSyncEvent(ev2);
```

Finally, an event dispatching sequence is done through:

```
eHandler.dispatchEvent(ev1);
Do SomethingElse();
eHandler.dispatchEvent(ev2);
```

If `eHandler` is a synchronization point the sequence on the reception of events will be:

```
Client1 receives ev1
Client2 receives ev1
SomethingElse is done
Client1 receives ev2
Client2 receives ev2
```

However, if `eHandler` is an AND synchronization point, dispatching `ev1` is delayed. Indeed, this event belongs to the same category as `ev2` (only their event source differ). Consequently, the sequence on the reception of events will be this time:

```
SomethingElse is done
Client1 receives ev1
Client2 receives ev1
Client1 receives ev2
Client2 receives ev2
```

This is the result of the fact that the "real" event propagation has to wait for the call:

```
eHandler.dispatchEvent(ev2);
```

before proceeding further.

The importance of synchronization points will become clear when event-based multimedia synchronization is addressed (see section 5.5).

### 5.4.2 Finite State Machines: Controller Objects

The notion of finite state machines is ubiquitous in computing, and it should not be necessary to define the notion here. Finite state machines (FSMs) are commonly used as elementary tools, e.g., to manage user interactions, to control communication among objects, etc. The behaviour of PREMO objects are often described in terms of states and state transitions, i.e., in terms of finite state machines. This usually provides a clear, and unambiguous specification. There is also a need to have finite state machines defined as separate object types. Multimedia systems are almost always interactive, and providing the elementary building blocks to create, e.g., interaction patterns is necessary for a proper specification of middleware.

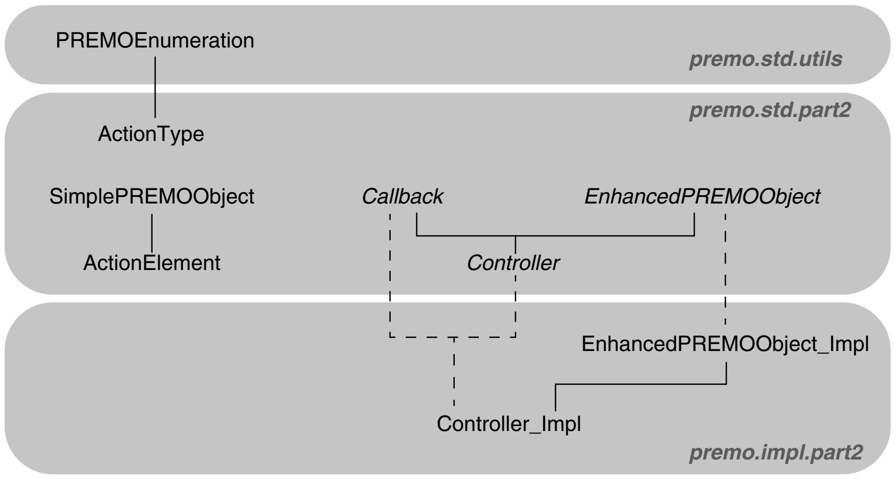


Figure 5-6 — PREMO Types Related to the `Controller` Object

The finite machines appearing in an interactive setting have some extra requirements. They must be:

- programmable, i.e., the end–user should have means to attach his/her own methods to each state transition of the object.
- monitorable, i.e., it should be possible to monitor state transitions from the outside easily, typically through some event propagation mechanism.

The `Controller` object type of PREMO provides these facilities. The programmability of the `Controller` objects is ensured by the specification of a number of protected methods as part of the object. The client of a `Controller` object does not (and should not) access these methods directly, because their invocation sequence is closely related to the state transition of the object. That is why they are protected. On the other hand, by defining a subtype for a `Controller` object, one can create specialized finite state machines where the required behaviour is coded into these protected methods.

The ability to monitor state transition is accomplished by dynamically attaching callback routines to state transitions. This means that if a state transition occurs, these callbacks are notified. For example, event handlers can be attached as callbacks to specific state transitions and, through these event handlers, any object which intends to monitor the state transitions can express its interest. Furthermore, `Controller` objects are defined as subtypes of `Callback`, too, which means that `Controller` objects can be chained together to form very complex interaction patterns.

#### 5.4.2.1 Detailed Specification of a `Controller`

*States* in `Controller` are represented by strings. The allowable states, as well as the current state of the object, can be retrieved by the operations:

```
public String  getCurrentState();
public String[]getPossibleStates();
```

These attributes are set during the initialization of the object (formally, this is done in subtypes. `Controller` itself is abstract). The attributes are “read–only”, i.e., the set of possible states cannot be changed by the client, nor can it directly change the current state.

The user can attach callbacks to various state transitions. This is done using a simple PREMO object:

```
public class ActionElement extends SimplePREMOObject {
   public Callback   eventHandler;
   public String     eventName;
}
```

describing the callback, and a PREMO enumeration of the form:

```
public final class ActionType
   extends premo.impl.utils.PREMOEnumeration {
      public static ActionType Enter;
      public static ActionType Leave;
}
```

The semantics of the `ActionElement` structure is clear: it holds the reference for the callback object which has to be notified of a state change. The `eventName` field is used to construct a suitable event structure. This event construction will be more thoroughly explained later.

Conceptually, each state may have two `ActionElement` instances assigned to them: one labelled as “enter” and the other labelled as “leave” action. Furthermore, each *pair* of states may refer to another instance of an `ActionElement`. These instances can be set and removed through the following methods (which are part of the public interface of `Controller`):

```
void setAction(String state, ActionElement action, ActionType aType)
   throws WrongState;
void removeAction(String state, ActionType aType)
   throws WrongState;
void setActionOnPair(String stateOld, String stateNew,
                     ActionElement action)
   throws WrongState;
void removeActionOnPair(String stateOld, String stateNew)
   throws WrongState;
```

The semantics of these routines is quite straightforward (the `WrongState` exception is raised if the required state is not a possible state for the `Controller` instance).

Finally, the last two methods of the public interface of `Controller` is:

```
void handleEvent(Event e);
void callback(Event e);
```

The two methods are completely identical; they are synonyms. This also meanst the the `handleEvent` operation is (just like callback) asynchronous. The operations trigger the state transition of the object. The new required state is the `eventName` field of the arguments (see page 66 for the specification of the `Event` structure). The `callback` method

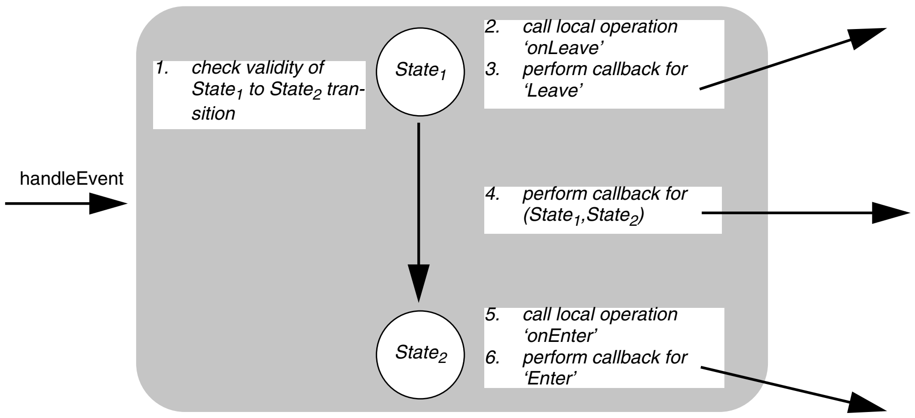

handleEvent
1. check validity of State$_1$ to State$_2$ transition
State$_1$
State$_2$
2. call local operation 'onLeave'
3. perform callback for 'Leave'
4. perform callback for (State$_1$,State$_2$)
5. call local operation 'onEnter'
6. perform callback for 'Enter'

Figure 5-7 — Behaviour of a `Controller` Object

makes the `Controller` object a `Callback` object, too. By declaring that `Controller` also extends the `Callback` interface, we ensure that the finite state machines can be chained through event propagation.

As said earlier, the exact semantics of the object depends on a set of protected operations, too (these are part of the implementation class of the `Controller`, i.e., `Controller_Impl`). These are as follows:

```
protected boolean checkTransition(Event e);
protected Event.EventData[]
   onLeave(Event e, String oldState, String newState);
protected Event.EventData[]
   onEnter(Event e, String oldState, String newState);
protected void handleUnknownEvent(Event e);
```

All elements are now in place for a precise specification of the object behaviour (see also Figure 5-7).

1. A state transition is requested by a client through the invocation of the operations `handleEvent` or `callback.` The event name is interpreted to be the state name to which the `Controller` object should transit. For the sake of this discussion, *State1* is the name of the current state of the object, and *State2* is the name of the requested state.

2. The controller object invokes the (protected) operation `checkTransition`, forwarding the argument of `handleEvent/callback`. This operation returns a boolean value indicating whether the transition is allowed. (By default, all transitions are allowed, i.e., `checkTransition` just tests whether the new state has been defined as a genuine state for the object. Subtypes may implement a more complex transition table by overriding this method).

3. If the transition is *not* allowed, the operation handleUnknownEvent is invoked, forwarding the argument of handleEvent, and handleEvent/callback finishes. handleUnknownEvent may decide to report an error, to forward the event to another Controller instance or an event handler, etc.
4. If the transition *is* allowed, the following steps are executed:
   - 4.1. The operation onLeave is invoked. This operation receives, as argument, the event structure appearing as the argument of handleEvent, as well as (the strings) *State1* and *State2*. The operation returns data suitable to be used as an event data tag in an event structure.
   - 4.2. If there is no ActionElement structure associated with *State1* labelled as "leave", this step is ignored, and go to 4.3 below. Otherwise, an event instance is created, using the event name in the ActionElement structure associated to *State1* labelled as "leave", and the event data returned by onLeave. A callback is then executed with the newly created event instance as argument.
   - 4.3. The value of the attribute denoting the current state is set to *State2*, i.e., the *real state transition occurs*.
   - 4.4. If an ActionElement is associated with the pair (*State1, State2*), a new event instance is created using the event name in the ActionElement and the array [*State1,State2*] as event data with the key "Transition". The callback operation on the Callback object, referenced by the ActionElement, is then invoked using this new event instance.
   - 4.5. The operation onEnter is invoked. This operation receives as arguments the event structure appearing as the argument of handleEvent, as well as (the strings) *State1* and *State2*. The operation returns data suitable to be used as an event data tag in an event structure.
   - 4.6. If there is no ActionElement structure associated to *State2*, and labelled as "enter", this step is ignored. Otherwise, an event instance is created, using the event name in the ActionElement structure associated to *State2* labelled as "enter", and the event data returned by onEnter. A callback is executed with the newly created event instance as argument.

The default behaviour of the onLeave and onEnter methods is to copy the data of the input event argument to the output. In other words, a Controller can simply forward the event data it receives without touching it.

The specification looks a bit complicated, but it is long rather than complex. Figure 5-7 shows the three possible places in a state transition where other Callback objects may be "hooked", i.e., which can be used to chain a Controller object into, e.g., an interaction pattern. The following example shows how a specialized FSM can be defined:

```
public class FSM extends Controller_Impl implements Controller
{
   // Initialization: fills up the state table
   public FSM() {
      super();
      possibleStates.addElement("First");
      possibleStates.addElement("Second");
      possibleStates.addElement("Third");
      currentState = "One";
   }
   // Specialize through the protected methods
   protected Event.EventData[]
      onLeave(Event e, String oldState,String newState)
   {
      System.out.println("Leave " + oldState + " for " + newState);
      // return the event data (using default action)
      return super.onLeave(e,oldState,newState);
   }
   protected Event.EventData[]
      onEnter(Event e,String oldState,String newState)
   {
      System.out.println("Arrive to " + newState +" from " + oldState);
      // return the event data (using default action):
      return super.onEnter(e,oldState,newState);
   }
}
```

The example is of course not very exciting, but it shows how FSMs can be defined in PREMO. Here is how an FSM instance can be monitored:

```
FSM fsm           = new FSM();
ActionElement act = new ActionElement();
act.eventHandler  = new MyCallback();
act.eventName     = "Monitor";
fsm.setAction( "Third", act, ActionType.Leave );
```

The effect of the code fragment is that any time the `FSM` object instance leaves the state "`Third`", the callback in `MyCallback` will be invoked with event whose name is "`Monitor`".

### 5.4.2.2 Activity of Controllers

Another issue, related to the `Controller` object, should be mentioned. As said in the introduction, controllers are frequently used as building blocks for interaction. It is therefore important that the state transition action would not suspend the caller of `handleEvent` for too long, i.e., this operation is asynchronous. This is also in line with the requirement on `Callback` objects, see page 68. Much like event handler objects (see page 76), the effect of `handleEvent` (and of `callback`) is to place the event instance into an internal event queue. A separate thread should be responsible for emptying this queue, possibly perform a state transition, and propagate the events through the callbacks.

### 5.4.3 Time Objects

#### 5.4.3.1 General Notions

Time is an essential notion in multimedia systems, and also one of the most difficult to grasp, primarily if full generality and distribution is to be taken into account. Although there are systems, special hardware equipment, etc., which are capable of providing precise control over time, these are rarely available in the type of computing environments in which PREMO is supposed to run. As a consequence, PREMO cannot include object specifications which rely on real–time control. In most of the cases implementation of such facilities would be impossible.

The pragmatic approach adopted by PREMO is to define a simple interface to time control, without *requiring* a certain accuracy. Instead, the local accuracy value can be inquired, and it is expected that the client would adapt its behaviour if the accuracy is, for example, reduced.

PREMO time objects may return elapsed time in different time units, ranging from picoseconds to years. Which unit to use is set by the client (a separate `TimeUnit` enumeration type is defined for that purpose). The accuracy of the time objects will, of course, depend on the current setting of the unit. Time objects may easily commit rounding errors and be off, for example, by one year if the unit is set to years. (For example, because our implementation relies on long integer values to measure time, it is not possible to express the value of 2.5 years!). On another extreme, it is very rare to have a computing environment which can provide an accurate measurement on the picosecond level. The typical case is that the various time objects would be reasonably accurate on the millisecond level, which is enough for multimedia presentation purposes. The current accuracy can be inquired by the client, and the unit used for the accuracy value can also be chosen. It is possible to measure the returned time value in years, but expect the corresponding accuracy value in milliseconds.

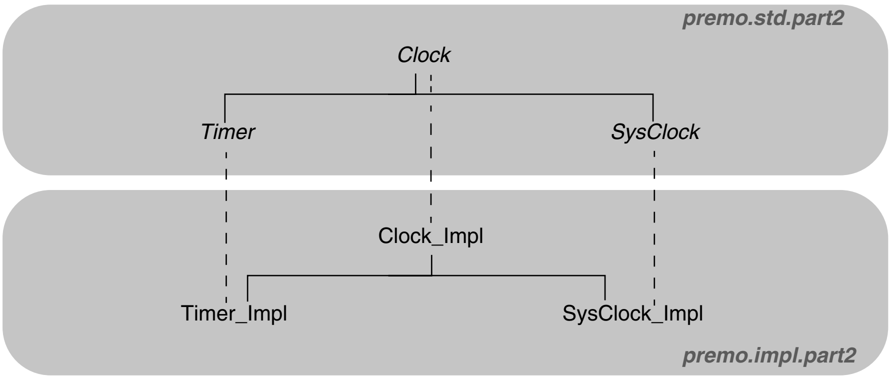


Figure 5-8 — Time Objects

On the abstract level, represented by the abstract type called `Clock`, elapsed time is measured as the returned value of an operation called `inquireTick`. Various subtypes of the `Clock` object attach a more detailed semantics to what the ticks really mean. This value and the accuracy obey the following relation. Suppose that the output of `inquireTick` is *T*, and the value of the accuracy is *A (both values are long integers in our case).* If the moment used by `inquireTick` as a starting point in time is *E* then, mathematically, the real actual time $T_r$, when `inquireTick` is called, follows the relation:

$$E + T - \frac{A}{2} \le T_r \le E + T + \frac{A}{2}$$

This also means that an accuracy of value zero represents the most accurate timing possible, and increasing values represent a loss in precision.[1]

[1] To be precise, this relation is valid if accuracy and time are measured using the same units. If this is not the case, *A* should be replaced by a function *f(A)*, which converts the accuracy value from its own units to the units used by *T*.

### 5.4.3.2 Specification of the PREMO Time Objects

The more formal specification of the time objects rely on the enumeration `TimeUnit`, which is defined as follows:

```
package premo.std.part2;
public final class TimeUnit
   extends premo.impl.utils.PREMOEnumeration {
   public static TimeUnit Picoseconds;
   public static TimeUnit Nanoseconds;
   public static TimeUnit Microseconds;
   public static TimeUnit Miliseconds;
   public static TimeUnit Second;
   public static TimeUnit Minute;
   public static TimeUnit Hour;
   public static TimeUnit Day;
   public static TimeUnit Month;
   public static TimeUnit Year;
}
```

which simply lists the various time units usable in PREMO. Using this enumeration, the abstract `Clock` interface is defined as follows:

```
package premo.std.part2;
public interface Clock
    extends EnhancedPREMOObject, java.rmi.Remote
{
   TimeUnit getTickUnit();
   void     setTickUnit(TimeUnit unit);
   TimeUnit getAccuracyUnit();
   void     setAccuracyUnit(TimeUnit unit);
   long     getAccuracy();
   long     inquireTick();
}
```

Based on the general description of the previous section, the semantics of each of these operations should be clear by now.

There are several objects in PREMO which implement this interface; two of them are usable by themselves, too. These are the system clock and the timer object.

The system clock object (called `SysClock`) provides real time information (modulo the accuracy of the clock, of course) to PREMO. `SysClock` does not add any new operations to its interface:

```
package premo.std.part2;
public interface SysClock extends Clock, java.rmi.Remote {
}
```

but defines the exact semantics of the `inquireTick` operation. It returns the elapsed time since 00:00AM, 1st of January 1970, UTC. For historical reasons, this starting point in time has been chosen by numerous computer systems (including Java), hence its choice in PREMO, too.

The `Timer` object models a stop–watch. It can be viewed as a tiny finite state machine, with three states: `TSTOPPED`, `TSTARTED`, and `TPAUSED` (these are static integer constants defined in a simple Java class called `State`) and managing an internal time

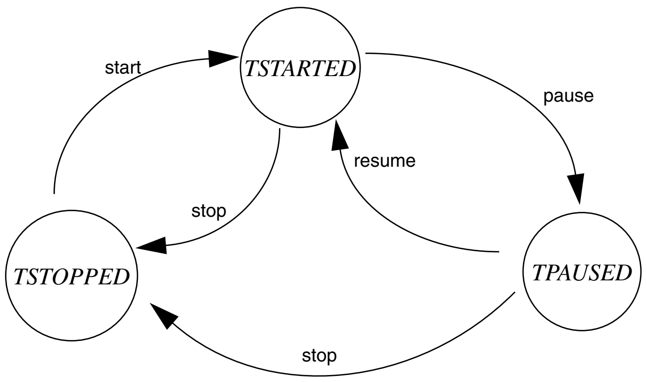


Figure 5-9 — State Transitions in a `Timer` Object

register. This register is reset to zero either when the object leaves the `TSTOPPED` state, or through an explicit `reset` operation. `inquireTick` returns the elapsed time the object spent in `TSTARTED` state since the last reset of the counter, but ignoring the time it spent in `TPAUSED` is ignored. The complete interface of the object is:

```
public interface Timer extends Clock,java.rmi.Remote
{
   int getTimerCurrentState();
   void start();
   void stop();
   void pause();
   void resume();
   void reset();
}
```

which include the obvious state transition operations and the explicit `reset`. Figure 5-9 shows the meaningful state transition operations. All other state transition calls (e.g., calling `pause` when the object is in the `TSTOPPED` state) are ignored.

## 5.5 Synchronization Facilities

One generally accepted and important characterization of multimedia systems is that they manage *continuous media* data. “This term refers to the temporal dimension of media, such as digital video and audio in that at the lowest level, the data are a sequence of samples — each with a time position. The timing constraints are enforced during playback or capture when the data are being viewed by humans.”[60] In some cases, such as animation and synthetic 3D sound, the samples may result from (sometimes complex) internal calculations (synthesis) whereas, in other cases, the samples are available through some data capture process.

Maintaining the presentation of a continuous media data stream at a sufficient rate and quality for human perception represents a significant challenge for multimedia systems, and may impose significant resource requirements on the multimedia computing environment. Aside from this inherent constraint (sometimes referred to as the problem of *intra–media synchronization*) a further difficulty arises from the fact that multimedia applications often wish to use *several* instances of continuous media data at the same time, such as an animation sequence with some accompanying sound or a video sequence with textual annotations. The difficulty here is that not only should the individual media data be presented with an acceptable quality, but well–defined portions of the various media content should appear, at least from a perceptual point of view, simultaneously; some parts of a sound track belong to a specific animation sequence, subtitles should appear with specified frames in a video sequence, etc. This problem is usually referred to as *inter–media* synchronization. The specific problems raised by intra–media synchronization is not addressed by PREMO, because this is media specific and falls outside the charter of a general reference model. In what follows, the term synchronization is always used to refer to inter–media synchronization.

Synchronization has received significant attention in the multimedia literature, see, for example, the book by Gibbs and Tsichritzis[34] or the survey paper Blakowki and Steinmetz[10] for further information and references on the topic. An efficient implementation of inter–media synchronization represents a major load on a multimedia system, and it is one of the major challenges in the field. What emerges from the experience of recent years is that, as is very often the case, one cannot pin down one specific place among all the computing layers (from hardware to the application) where the synchronization problem should be solved. Instead, the requirements of synchronization should be considered across all layers, i.e., in network technology, operating systems, software architectures, programming languages, etc. and user interfaces.

The synchronization facilities of PREMO concentrate on one aspect of a complete solution, namely, on a conceptual model and software architecture aimed at inter–media synchronization. It provides general facilities which can be used to *implement* various synchronization specifications which use interval–based, axes–based, or other declarative methods (see again, e.g., the survey paper in [10] for further details and references). In line with the middleware nature of PREMO, the goal is to provide a general mechanism upon which these various declarations can be implemented, instead of dictating one specific approach to be used for synchronization.

The PREMO synchronization model is based on the fact that objects in PREMO are active. Different continuous media (e.g., a video sequence and corresponding sound track) are modelled as concurrent activities that may have to reach specific milestones at distinct and possibly user definable synchronization points. This is the *event–based* synchronization approach, which forms the basic layer of synchronization in PREMO. Although a large number of synchronization tasks are, in practice, related to synchronization in time, the choice of an essentially "timeless" synchronization scheme as a basis offers greater flexibility. While time–related synchronization schemes can be built on top of an event–based synchronization model, it is sometimes necessary to support purely event–based synchronization to achieve special effects required by some application (see, for example, the application described on page 99).

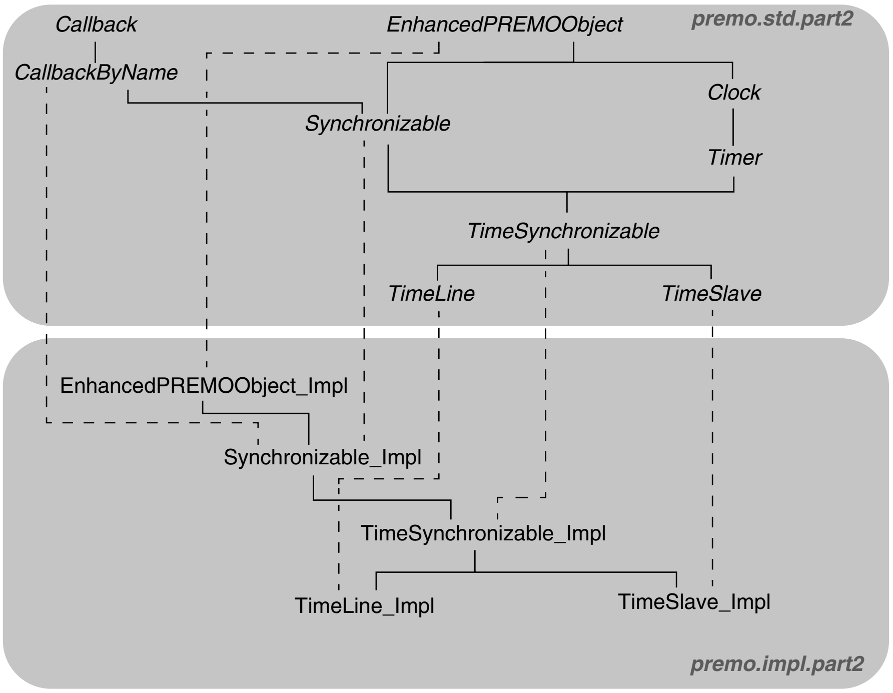


Figure 5-10 — Objects for Synchronization

In line with the object–oriented approach of PREMO, the synchronization model uses abstract object types that capture the essential features of synchronization. Some of them have already been presented in earlier chapters, whereas some are more specifically tailored at synchronization. These are:

- *synchronizable objects*, and their various subtypes (see Figure 5-10), to be presented in more detailed in this section.
- *synchronization points*, (and event handlers in general), see section 5.4.1.3.
- *time objects*, see section 5.4.3.

Among all these, the various synchronizable objects are by far the most complex ones. The rest of this section concentrates on their detailed specification.

## 5.5.1 Synchronizable Objects

### 5.5.1.1 Overview: Event–Based Synchronization

Synchronizable objects in PREMO provide a high–level abstraction for media data presentation and synchronization. Media datum is taken here in a very abstract sense, and it is only the various subtypes which attach specific semantics to it. On this abstract level, synchronizable objects have an internal progression along an internal, one dimensional coordinate space, also referred to as *progression space*. The active nature of the synchronizable objects play a paramount importance. Synchronizable objects progress along their progression spaces independently from one another, using their own thread of control (their "virtual processor").

The progression space can be represented by integers, doubles, or long integers. To be more precise, the progression space is, conceptually, one of:

$$double_{\infty} == double \cup \{-\infty, \infty\}$$
$$int_{\infty} == int \cup \{-\infty, \infty\}$$
$$long_{\infty} == long \cup \{-\infty, \infty\}$$

i.e., the concepts of positive and negative "infinity" are also meaningful. The obvious extension of the notions "greater than", "smaller than", etc., on these spaces allows the behaviour of synchronizable objects to be defined more succinctly. We define no special Java classes to represent these types. We will simply identify the `MAX_VALUE` and the `MIN_VALUE` constants of the `Integer`, `Double`, and `Long` Java classes with their respective infinity values[1]. It is up to the implementation of the synchronizable objects to manage the arithmetic properly. To simplify the discussion, the symbol *"C"* will be used in this section to denote this progression space. Subtypes of synchronizable objects specify the exact numeric type being used and add a semantic meaning to this coordinate space. Attributes of the progression, such as span (the interval of interest within this coordinate space), can be set through operations defined on the synchronizable object.

The progression space[2] can be used to describe various types of progression. For example, media objects may represent time, video frame numbers along this space, animation frame numbers, sound samples, etc. The choice of the semantic content of the progression space may also depend on the application. For example, if the object represents a symphony, the progression space units may represent either the various movements of the symphony, or the bars within a movement, or the notes themselves, or even the individual musical samples. The choice will obviously depend on the level of synchronization granularity the application requires.

*Reference points* are points on the internal coordinate space of synchronizable objects where *synchronization elements* can be attached (see also Figure 5-11). Synchronization elements are structures which contain a reference to in an event instance, a reference to a PREMO `Callback` object, and, finally, a boolean `Wait` flag. When progression goes over a reference point, the synchronizable object makes a call to the `callback` operation of the object stored in the reference point, using the stored event instance as an argument to the call. Then, it suspends itself if the `Wait` flag is set to `true`, or continues progression otherwise. Through this mechanism, the synchronizable object

[1] Note that the PREMO document defines these types as separate non–object types. The rich facilities provided by the Java `Integer`, `Double`, and `Long` classes makes it unnecessary to define separate types for the Java version.

[2] The multimedia literature also uses the term *LDU* (for Logical Data Units) to denote the same concept[10].

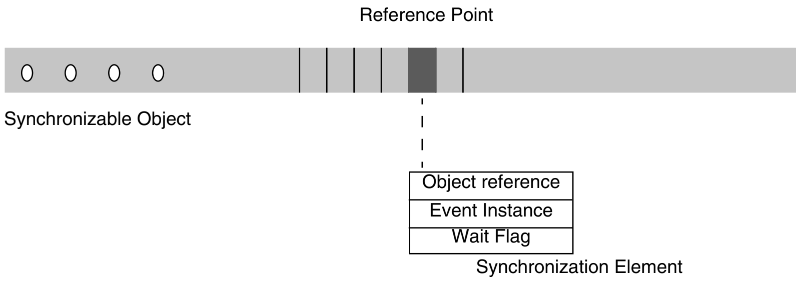


Figure 5-11 — A Synchronizable Object

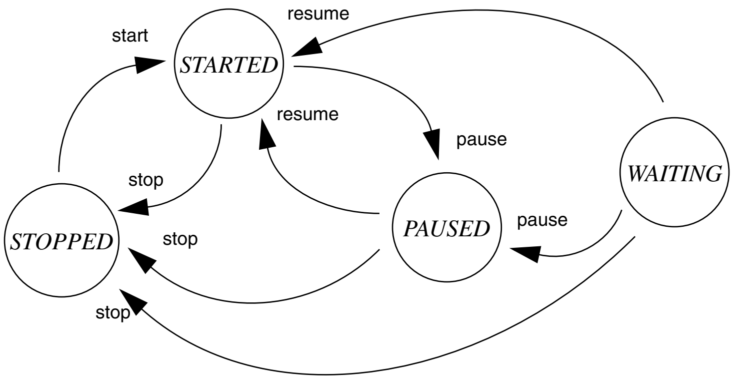


Figure 5-12 — State Transitions of a Synchronizable Object

can stop other objects, restart them, suspend them, etc. Operations are defined on synchronizable objects to add or delete reference points with their related synchronization elements.

In more precise terms, a `Synchronizable` object type is defined in PREMO as a supertype for all objects which may be subject to synchronization. This object is defined to be a finite state machine. The possible states, the major state transitions, and the operations resulting in state transitions, are shown in Figure 5-12. The initial state is `STOPPED`. Note that no operation is defined for a transition into state `WAITING`. The only way a `Synchronizable` object can go into the `WAITING` state is through its internal processing cycle (see below).

The object maintains a current position on its progression space. The progression of the object along its internal coordinate space happens within a (possibly infinite) interval of this space, called the *span,* defined through a start and an end point. If the object's state is `STARTED`, the object carries out its internal processing in a loop of processing stages. Each stage consists of the following steps:

1. The value of the current position is advanced using a (protected) operation `progressPosition` (defined as part of the object's specification) which returns the required next position.
2. This required position is compared with the current position and the end position, and the following actions are performed:

   2.1. If there are reference points lying between the current position and the newly calculated position (or there is a reference point set at the new position), then any associated synchronization actions are performed (in the order in which they are defined on *C*). This means:

   – Perform data presentation for any data identified by the points on the progression space between the current position or the previous reference point and the next reference point (or the end point). Formally, this is again done through the invocation of a protected operation, called `processData`.

   – Invoke the `callback` operation on the `Callback` object, whose reference is stored in the reference point, using the stored event as an argument.

   – If the `Wait` flag stored in the synchronization element belonging to the reference point is set to `true`, the object's state is changed to `WAITING`. If the state of the object is set back, eventually, to `STARTED`, the stage continues at this point.

   2.2. If the required position is smaller than the end position, then this becomes the local position and the processing stage is finished.

While in `PAUSED` or `WAITING` state, the object can only react to a very restricted set of operation requests. The attributes of the object may be retrieved (but not set) and the `resume` or `stop` operations may be invoked, which may result in a change in state. The difference between `PAUSED` and `WAITING` is that, in the latter case, the object returns to the place where it had been suspended by a `Wait` flag, whereas, in the former case, a complete new processing stage begins. In other words, no new call to `progressPosition` occurs when returning from `WAITING` state. The differentiation between these two states, i.e., the use of the `Wait` flag, is essential. This mechanism ensures an instantaneous control over the behaviour of the object at a synchronization point. If the object could only be stopped by another object via a `pause` call, an unwanted race condition could occur.

This description refers to the situation where the direction of progression is identical to the inherent ordering of C. The direction can also be reversed by the client. This means that the roles of the span's start and end points are reversed, as well as the order with which the reference points and the data presentation are considered.

What happens if the progression reaches the end of the span? Attributes are defined which control whether the object should loop at that stage or not. There are two such controlling attributes: a `repeatFlag`, which is a boolean, and a `NLoop` value, which is a positive integer. When the required new position is greater or equal to the end of the span, this is how the object behaves:

- If the `repeatFlag` is `true`, this means that the object repeats playing over the span indefinitely while in `STARTED` state. In other words, the current position value is set to the start of the span (or the end of the span if progression is backward) and the next processing stage starts there.

- If the flag is set to `false`, the number of loops the object has to perform is controlled by the `nloop` value. The object maintains an internal loop counter (whose value can be inquired, by the way), which is initialized to the value of `nloop` when the object enters the `STARTED` state from `STOPPED`. This counter represents the number of times the object has to play the defined span. When the required number of loops have been played, the synchronizable object automatically changes its state to `STOPPED`.

Note that two aspects of this specification are left hitherto unspecified in the definition of `Synchronizable`:

- what "data presentation" means (i.e., the semantics of the `processData` operation), and
- what it means to progress through the reference points (i.e., the semantics of the `progressPosition` operation).

Both these aspects should be specified in the subtypes of `Synchronizable`. The abstract specification of a synchronizable object is such that no media specific semantics are directly attached to it. Subtypes, realizing specific media control should, through specialization, attach semantics to the object through their choice of the type of the internal coordinate system, through a proper specification of the `processData` and the `progressPosition` operations. Processing data may mean, in simple cases, to present the data on a screen (e.g., to put the video frames into a window). In more complicated situations, however, processing the data may involve the control of other devices, either software or hardware. The following chapters will elaborate on this aspect further. The operation `progressPosition` defines what it really means to "advance" along the internal coordinate system and also controls the granularity of progression. For example, this progression may mean the generation of the next animation frame, decoding the next video frame, advance in time, etc. A good example for the roles of `processData` and `progressPosition` is the way a simple MPEG decoder could work in this setting. The `progressPosition` operation can successively return the frame positions corresponding to the key frames ("I–frames", in the MPEG specification) in the MPEG encoding process. Indeed, key frames represent the natural points where an MPEG object could stop for synchronization in combination with its procedure to fetch data. Data processing may then mean to display the frames between two keyframes.

The "target" object in the synchronization element (i.e., the object which is notified that the synchronizable object has reached a reference point) can be any PREMO `Callback` object. This means that, for example, a synchronizable object can be combined with various event handlers or controller objects; furthermore, the `Synchronizable` object type is also defined to implement the `Callback` interface, i.e., the `Synchronizable` objects can also notify one another when crossing reference points. The combination of all these objects in one synchronization pattern offers very rich facilities.

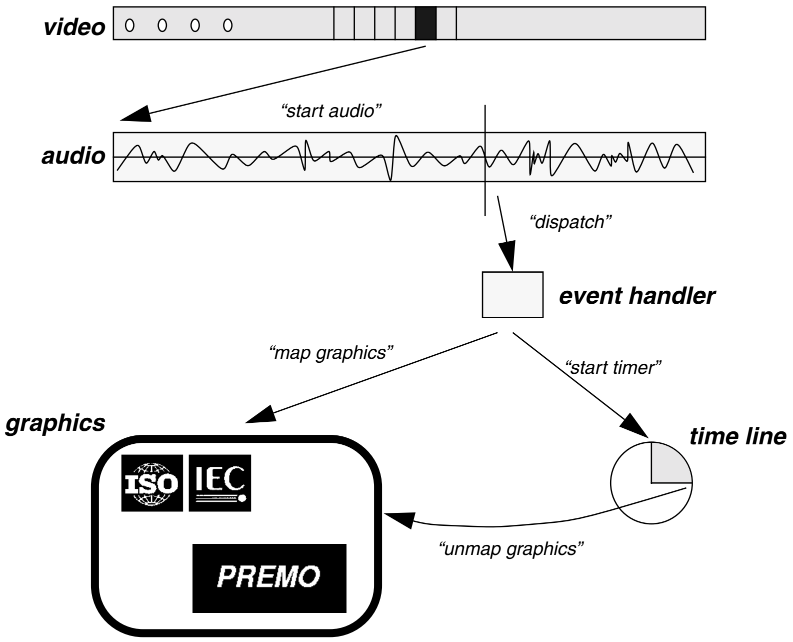


Figure 5-13 — A Simple Example for the Use of Synchronizable Objects

Figure 5-13 shows a very simple example using the synchronization mechanism described above. The three media objects (video, audio, and graphics) in the figure are subtypes of `Synchronizable`, as is the time line object. They all add specific semantics to this supertype. Reference points and synchronization points are set up for the objects. The name of the operations eventually used through the `callback` operations are denoted on the figure. The effective synchronization pattern is:

1. The video starts to play; when it reaches its reference point, it sends a message to the audio object. The video object then continues to progress.
2. As a result of the message received from the video object, the audio object begins to play (in parallel with the video object). When it reaches its reference point, it sends a message to both the graphics and the time line objects (the role of the event handler object is to dispatch the same event among several targets). The audio object then continues to progress.
3. The graphics appears on the screen and, in parallel, a timer begins to tick. The timer has its own reference point set to, e.g., 15 seconds; when this reference point is reached, the message "unmap graphics" is sent to the graphics media object, which unmaps the image from the screen.

Although this example is obviously a simplified one, it illustrates the main mechanism at work when event–based synchronization is used.

The reader may wonder why PREMO chose the event–based synchronization approach as its basic model; most of the synchronization literature and practice deals with time–based synchronization only, i.e., where synchronization is expressed in terms of milliseconds, time intervals, etc. The reason is that there are important multimedia applications where the purely time–based notions break down, where time does not really make sense. A real–life example is as follows. The application involves so–called *cineloops*, which are a movie–like representation of a sequence of ultrasound scans made for medical purposes. Details of how these cineloops are created are not of real interest here. They should be considered as special media objects which behave much like a sequence of images. Figure 5-14 shows a screen snapshot involving two cineloops, taken by scanning the human heart. When a cineloop is recorded, one will usually also record an ECG trace (an electrocardiogram), shown below the cineloop images. Also, in many cases, it is useful to compare cineloops recorded under different conditions or at different times. For example, a stress test compares the movement of the heart wall after a rest and just after the person has exercised, when the heart rate is much higher. Figure 5-14 shows two such recordings, each with its corresponding ECG trace.

The difficulty of playback, when these cineloops are used, is that physicians want to see side by side a particular event of the heart beat, e.g., the start or the end of a contraction of a heart chamber. They are not really interested in the *timing* of the movements. However, the different phases of movement that constitute a heartbeat do not speed up with the same rate when the heart beats faster, i.e., it is not sufficient to just speed up or slow down the cineloops. In other words, synchronizing between the two cineloops cannot be done in terms of time. Indeed, the notion of time does not really make any sense in this particular case.

The synchronization problem can be solved if event–based approach is used. Synchronization elements are set against reference points representing the required events in the cineloop (or can be set on the ECG playback, and have a close synchronization of a cineloop with its corresponding ECG), and the playback can be synchronized within a framework using synchronizable objects and synchronization points. Details of how this can be done can be found in the paper of Lie and Correia[63], which uses the MADE toolkit[42] for this purpose in a real–life medical application involving such cineloops (this toolkit uses a very similar synchronization mechanism to the one proposed by PREMO).

Similar examples can also be found in other application areas, e.g., computer animation, virtual reality, etc. Of course, time–based synchronization is extremely important, and subtypes of the `Synchronizable` objects are defined in PREMO for that purpose (see section 5.5.2 below) but choosing event–based synchronization gives the generality which is necessary for a standard reference model for multimedia applications.

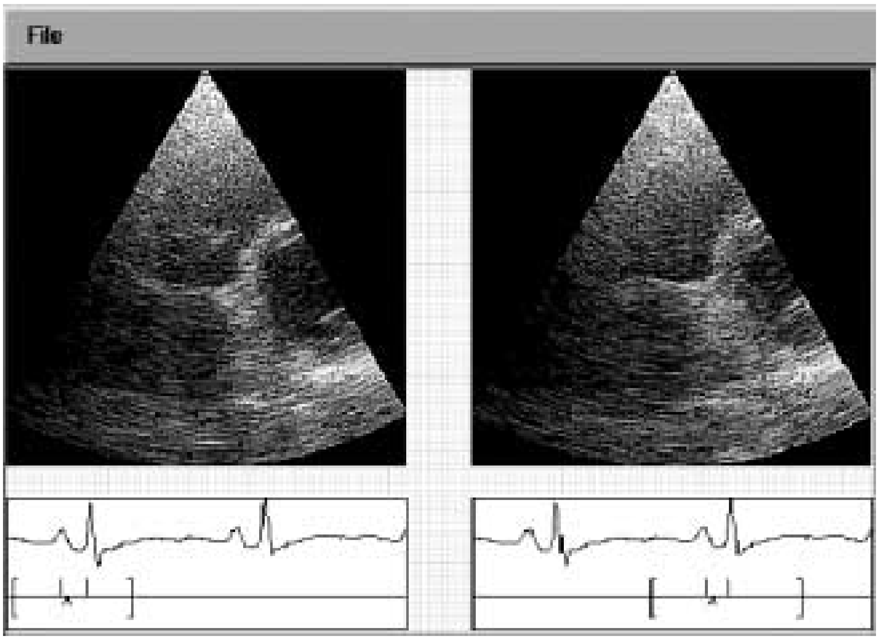


Figure 5-14 — A Cineloop Example

### 5.5.1.2 State Transition Monitoring

The mechanism described in the previous allows various PREMO objects to monitor media progress. This can be done in setting the appropriate reference points and (for example) registering to the event handlers which might handle the events coming from that synchronizable object.

Independently from media progress, however, objects may also wish to monitor the state transition of the `Synchronizable` object. For this purpose, much like in the case of `Controller` objects, references to `Callback` objects can be assigned to pairs of states; whenever the `Synchronizable` object makes a state transition which corresponds to this pair, the `callback` operation on the registered object will be invoked.

### 5.5.1.3 Detailed Specification of the `Synchronizable` Object

The detailed (Java) specification of the `Synchronizable` object is not very complex, once the underlying principles are understood. The only slightly complicated feature is how the exact type of the progression space is defined.

Remember that the progression space of a `Synchronizable` object can be based on double, integer, or long integer values (see page 92). The choice of which type is being used is the subtype's, and it depends on the semantics of the media which is processed. The `Synchronizable` object is abstract (i.e., it cannot be instantiated per se), and subtypes have the ability to fix the type through, e.g., their constructors.

In the Java interface specification, the (standard) Java class `Number` is used to access points on the progression space. This class is a common superclass of the `Double`, `Int`, and `Long` Java classes, that is it can provide the necessary generality and abstraction level. There is, however, one complication: for one specific `Synchronizable` *instance*, the type to be used for its progression space must be the same for all calls, and this must be checked for *all* method invocations which involve a progression space value. In our interface specification and implementation, all these methods may therefore throw the standard Java `IllegalArgumentException` in case the exact type of the `Number` argument does not match the type of the progression space.[1)]

The `Synchronizable` object interface has the following inheritance structure:

```
public interface Synchronizable
    extends CallbackByName, EnhancedPREMOObject, java.rmi.Remote
```

The only important point is that it is a subtype of `CallbackByName`, too. This means that it is a `Callback`, and that the semantics of the `callback` operation is such that it would use the name of the event in its argument to find the operation to be invoked internally (see section 5.3.3). This means that, if `Synchronizable` objects are bound together in various patterns, operations such as `stop`, `pause`, etc., can be coded into the synchronization elements easily.

The methods defined in the `Synchronizable` interface can be grouped in various categories. This categorization will be used in what follows to gain a better overview of the methods.

#### *5.5.1.3.1 Retrieve only Attributes*

`Synchronizable` objects define a number of internal attributes which are retrieve only, i.e., they are either constants, or they can be changed only by the object itself. They are all related to the internal processing loop of the object.

```
int getCurrentState() throws java.rmi.RemoteException;
```

This operation returns the state of the object. The integer codes for the states are grouped as final static integer values of the class `State`. The relevant constants are:

```
public static final int STOPPED  = 0;
public static final int STARTED  = 1;
public static final int PAUSED   = 2;
public static final int WAITING  = 3;
```

The operations

[1)] For C++ connoisseurs: the original ISO PREMO document makes use of a template class like mechanism at this point, where the type parameter makes the exact choice for the progression space type. In other words, the type would then be checked automatically. However, Java does not have any templates, so an explicit check, with a corresponding exception, had to substitute the template specification.

```
Number getMinimumPosition();
Number getMaximumPosition();
```

return the absolute minimal and maximal positions for the given instance. All position values, when using this object instance, must be in this interval. The values `MAX_VALUE` or `MIN_VALUE` may also be returned, denoting positive, or negative, infinity, respectively.

The operation

```
Number getCurrentPosition();
```

returns, as its name suggests, the current position value, while the operation

```
int getLoopCounter();
```

returns the current loop counter value. Both these values change while the object progresses in its processing loop.

##### *5.5.1.3.2 Settable Attributes*

These attributes control and/or modify the progression of the object along the progression space. All of them have in common that they can be set only if the object is in `STOPPED` state, although they can be retrieved at any time. The exception `WrongState` is thrown if an attempt is made to change these values in a wrong state. The semantics of all these attributes must be clear by now.

```
void setDirection(Direction where) throwsWrongState;
Direction getDirection()

void setStartPosition(Number position)
   throws   WrongValue, WrongState, IllegalArgumentException;
Number getStartPosition()

void setEndPosition(Number position)
   throws   WrongValue, WrongState, IllegalArgumentException;
Number getEndPosition()

void setRepeatFlag(boolean flag) throwsWrongState;
boolean getRepeatFlag();

void setNLoop(int value)throws WrongState,IllegalArgumentException;
int getNLoop();

void resetLoopCounter() throws WrongState;
```

The operation `resetLoopCounter` sets the loop counter value back to `NLoop`, i.e., all the loops will start all over again. `Direction`, which is used in some of the operations above, is a simple enumeration:

```
public final class Direction extends premo.impl.utils.PREMOEnumeration
{
    public static Direction Forward;
    public static Direction Backward;
}
```

One more operation should be listed in this category, insofar as it modifies the current position value:

```
void jump(Number position)
   throws   WrongState, WrongValue, IllegalArgumentException;
```

The difference is that this operation can also be issued when the object is in `PAUSED` state and not only in `STOPPED` state.

#### *5.5.1.3.3 Management of Reference Points*

Operations in this category can add or delete reference points and their corresponding synchronization element. There are basically two ways reference points can be defined:

- specifying a synchronization element at an absolute position, or
- specifying a repeated (periodic) synchronization element.

The second possibility means that the same synchronization element is (conceptually) repeated at the points:

$$startRefPoint + p, startRefPoint + 2 \cdot p, \ldots, startRefPoint + n \cdot p$$

until an *endRefPoint* value is exceeded. The value of *p* is the *periodicity* of the synchronization element.

Reference points can be set or deleted only when the object is in `STOPPED` or `PAUSED` state. The `WrongState` exception is thrown in all other cases. All operations make use of the simple PREMO object:

```
public class SyncElement extends SimplePREMOObject {
    public Callback  eventHandler;
    public Event     syncEvent;
    public boolean   waitFlag;
}
```

to describe a synchronization element. The operations themselves are as follows:

```
void setSyncElement(Number position, SyncElement syncElement)
   throws   WrongState, WrongValue, IllegalArgumentException;
void deleteSyncElement(Number position)
   throws   WrongState, WrongValue, IllegalArgumentException;

void setPeriodicSyncElement( Number startRefPoint,Number endRefPoint,
                             Number periodicity,
                             SyncElement syncData)
   throws   WrongState, WrongValue, IllegalArgumentException;
void deletePeriodicSyncElement( Number startRefPoint,
                                Number endRefPoint,
                                Number periodicity)
   throws   WrongState, WrongValue, IllegalArgumentException;
```

A separate operation is defined to inquire the synchronization elements in a specific interval. This operation can be invoked at any time (note the use of an inner class in the specification):

```
public class SyncInfo {
   public SyncElement   syncElement;
   public Number        position;
}
public SyncInfo[] getSyncElements(Number posMin, Number posMax)
   throws   WrongValue, IllegalArgumentException;
```

#### *5.5.1.3.4 Management of Action Elements*

The following two operations are related to state transition monitoring actions. They set and delete an action element to a pair of allowable states. An action element is identical to the structure used for `Controller` objects:

```
public class ActionElement extends SimplePREMOObject {
   public Callback   eventHandler;
   public String     eventName;
}
```

Using this action element means creating an Event instance with the specific name. The event data (which is a key–value pair) uses the key "`Transition`" the pair of states, which has just been used for transition, as value.

The operations themselves are quite simple; just as in the case of reference point management, they can be invoked in `STOPPED` or `PAUSED` states only.

```
void setActionOnPair(int stateOld, int stateNew, ActionElement action)
   throws   WrongState;
void removeActionOnPair(int stateOld, int stateNew)
   throws   WrongState;
```

Again, a `WrongState` exception is thrown if one of the arguments does not refer to a valid state.

#### *5.5.1.3.5 General Reset*

This category has only one operation:

```
void clearSyncElements() throws WrongState;
```

which deletes *all* reference points and action elements. It can be issued in `STOPPED` or `PAUSED` states only, of course.

### 5.5.1.4 `Synchronizable` Objects as Callbacks

A `Synchronizable` object has been defined as a subtype of `CallbackByName`. This means that other objects (most often other `Synchronizable` objects) may call the various state transition operations, such as `start`, `resume`, etc., through their respective reference points. This is done by simply setting the name of the event being used in the callback to the right call.

There is, however, a subtle but very important difference between calling these operations directly, or doing it indirectly, through the `callback` method. All operations on `Synchronizable` are defined as synchronous, i.e., the caller of the operation will be suspended as long as the operation is not performed. By contrast, however, `callback` is essentially asynchronous (see section 5.3.3), i.e., the caller is *not* suspended when the state transition operation is invoked through this mechanism. This detail is very important in practice; if this was not the case then, for example, the video object on Figure 5-13 (see page 97) would be unnecessarily suspended while starting the audio.

## 5.5.2 Time and Synchronizable Objects

The synchronization model presented in section 5.5.1 is event–based, i.e., the notion of time is not part of the abstraction level in that model. Clearly, applications also require a more elaborate version, which would allow them to reason with time. This is achieved in PREMO through the specification of a separate hierarchy of time objects and the specialization of a basic synchronizable object which would include the notion of time. The time objects have already been presented in section 5.4.3. This section presents how PREMO combines the notion of time with that of synchronization.

The combination of the two facilities is embodied in the `TimeSynchronizable` object type, which implements both the `Timer` interface (see page 90) and the `Synchronizable` interface, as defined earlier in this section. In other words, a `TimeSynchronizable` has its own progression space with its own autonomous progression, as well as being essentially a stop–watch with its own notion of time.

Of course, implementing both interfaces is not enough; the semantics of the two object types have to be reconciled. The semantic issues which arise are:

- How do the two finite state machines, represented by the two supertypes, coexist?
- What is the relationship of the progression space and the clock?
- How can the client control progression data in terms of time?

Let us take these questions one by one.

### 5.5.2.1 Stop–Watch and Progression

The `Timer` object is a finite state machine, whose state transitions are represented on Figure 5-9 on page 90. The `Synchronizable` object's state transitions are depicted on Figure 5-12 on page 94. The two state machines are very similar to one another. The most noticeable difference is that the `Synchronizable` state machine has one extra state (`WAITING`), with some extra state transition operations. A straightforward way of defining a "derived" state machine for the `TimeSynchronizable` object is to adopt the state machine of `Synchronizable`, and identify the states of the stop–watch with states of the `Synchronizable`. This is shown on Figure 5-15. What this merger means, semantically, is that

- while the object progresses along its progression space, the clock is running, too;
- "pausing" the progression means pausing the clock as well; also, if the object is waiting on a reference point, the clock is paused;
- stopping progression means stopping and resetting the stop–watch, too.

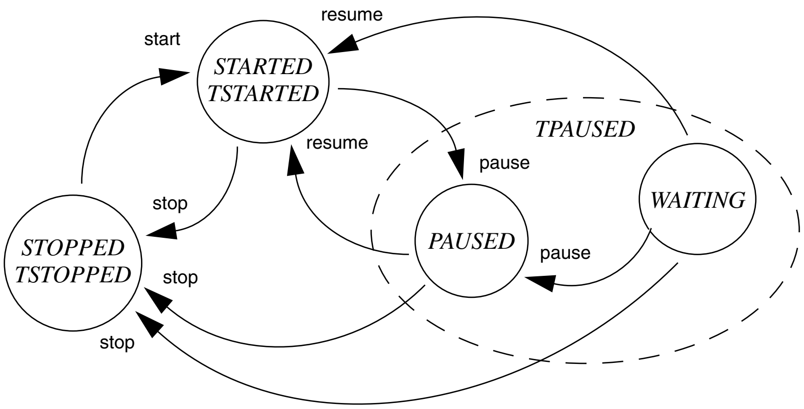


Figure 5-15 — State Transitions of a Time Synchronizable Object

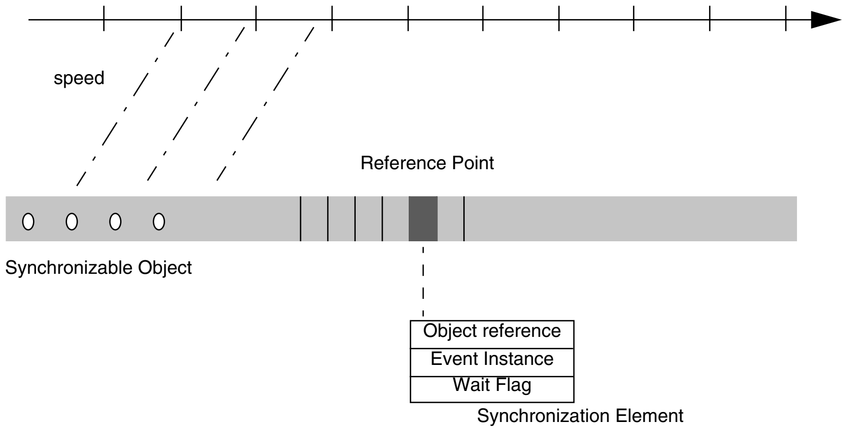


Figure 5-16 — A Time Synchronizable Object

The effects of the various state transition operations, inherited both from `Timer` and `Synchronizable` are clear from the figure; resuming a clock and resuming progression is done simultaneously, etc.

### 5.5.2.2 Time and Progression Space

Merging the state machines does not yet create a link between two "progressions", namely moving along the progression space and along time. This link is created by an additional attribute, called the *speed* (see Figure 5-16). As its name suggests, this attribute expresses the ratio between time and moving along the progression space. It is the number of units on the progression space per clock tick. It must be emphasized, that the value of the speed depends on the units used to measure time, settable through the `setTimeUnit` operations (inherited from the `Timer`).

In general, the value of speed can be set and retrieved through the operations:

```
void setSpeed(double speed) throwsWrongState;
double getSpeed();
```

These operations are defined as part of the `TimeSynchronizable` interface (setting speed is possible in `STOPPED` state only, hence the `WrongState` exception in the specification). However, subtypes of `TimeSynchronizable` may not have a settable speed (see the `TimeLine` object below, for example) in which case the `setSpeed` operation is without effect. Most of the `TimeSynchronizable` objects do allow a variable speed, which can be used to speed up or slow down sound rates, animation frame rates, etc.

#### 5.5.2.3 Reference Point Specifications in Time

The `TimeSynchronizable` object inherits a time register from `Timer`, too. In a `Timer` object, this register is reset to zero either when the object leaves the `TSTOPPED` state, or through an explicit `reset` operation. In the case of the `TimeSynchronizable` object, resetting this time register has additional semantics; it also puts a conceptual marker against the current position on the progression space as well as resetting the time register[1)]. This marked position on the progression space and the speed value together define a linear transformation between the two spaces which makes an identification of time and progression possible.

`TimeSynchronizable` defines two operations which can be used to convert progression space values into time and vice versa. These operations are:

```
Number   timeToSpace(long time)
long     spaceToTime(Number position) throwsIllegalArgumentException;
```

Using these two conversion methods, the client can choose to define reference points in time, converting the time values into position values, and use the inherited reference point management operations to set the reference points.

PREMO also defines a set of operations for `TimeSynchronizable`, which are the semantic equivalents of the operations defined in sections 5.5.1.3.1, 5.5.1.3.2, and 5.5.1.3.3, with the notable exception that the type `long` is used instead of `Number`. Their semantics is identical to their `Synchronizable` counterpart, except that positioning is done in time and the current linear transformation between time and progression space is used to identify the active reference points. The operations are (for simplicity, the exceptions are not listed this time):

1) Note that, if the register is reset through leaving the `TSTOPPED` state, this position is the default start postion of the object.

```
long getTimeCurrentPosition();
long getTimeMinimumPosition();
long getTimeMaximumPosition();
void setTimeStartPosition(long position);
long getTimeStartPosition();
void setTimeEndPosition(long position);
long getTimeEndPosition();
void jump(long position);
void setSyncElement(long position, SyncElement syncElement);
void deleteSyncElement(long position);
public class TimeSyncInfo {
   public SyncElement   syncElement;
   public long          position;
}
TimeSyncInfo[] getSyncElements(long posMin, long posMax)
void setPeriodicSyncElement( long startRefPoint, long endRefPoint,
                             long periodicity,
                             SyncElement syncData)
void deletePeriodicSyncElement( long startRefPoint, long endRefPoint,
                                long periodicity)
```

Using these methods the client can, for example, make a jump in time directly, without using the progression space.

There may, however, be a difference between using these operations explicitly, or setting a synchronization element through the use of `timeToSpace`. Synchronization elements defined in terms of time are also stored internally in terms of time. This becomes important if the client resets the time register through an explicit `reset` operation. Indeed, resetting the time register means changing the linear transformation connecting time to progression space; consequently, if this reset is done while “playing”, some reference points may, for example, become “active” again. Of course, if an explicit `reset` is never used, i.e., the linear transformation changes only through state transition from `STOPPED`, use of time or use of the progression space becomes synonymous.

It must be emphasized that the abstract `TimeSynchronizable` object still leaves a lot of implementation details for the specific subtypes of this object. How the progression in time and progression in “space” are related to one another is often media specific. To take a simple example, let us consider at a (subtype of) `TimeSynchronizable` managing a video frame sequence. When asked for the next frame (i.e., when the `progressPosition` operation is invoked), the object has to consider the local clock, the value of `speed` and, maybe, some specifics of the video frame sequence, before deciding on the next “current” position. Sophisticated implementations of PREMO, i.e., of the `TimeSynchronizable` object, might decide to generalize such mechanisms, thereby simplifying the task of the implementors of specific devices, but PREMO does not specify these details.

The `TimeSynchronizable` object has all the usual synchronization features attached by various multimedia systems to their basic media representation. However, in most of the systems, the distinction between (relative) time and the internal progression space (e.g., video frames) is blurred, usually in favour of time only. PREMO maintains this dual nature of media data, and leaves it to applications to decide which aspect of media

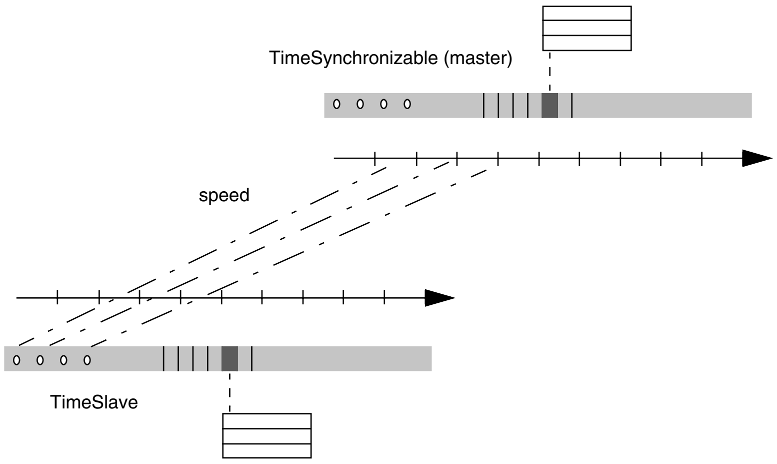


Figure 5-17 — A Time slave Object

behaviour is more relevant in a concrete synchronization setting. This separation is one of the advantages of a clear object oriented specification offered by a standard such as PREMO

## 5.5.3 Combining `TimeSynchronizable` Objects: Time Slaves

`TimeSynchronizable` objects are appropriate for creating complex synchronization patterns involving time. In an ideal world, where all local timers would represent an absolutely precise real time, this would be enough. However, multimedia systems rarely operate in an ideal world, and in practice all local timers will have a slightly different speed, accuracy, etc. Hence the necessity of implementing mechanisms which may monitor possible discrepancies.

PREMO does not aim at offering a full solution for this problem, because the necessary reactions, the tolerated discrepancies, etc., are usually application dependent. PREMO defines the basic mechanism which allows applications to implement a specific behaviour, and it does this in terms of a new object type, called a `TimeSlave` object. What this object essentially does is to control its own behaviour in terms of the timer data of *another* `TimeSynchronizable` object..

As we saw in section 5.5.2.2, the notion of speed is the essential attribute which binds the progression space of a `TimeSynchronizable` object and its own clock. The `TimeSlave` object is a subtype of `TimeSynchronizable`, which allows the client to set a *master* `TimeSynchronizable` object:

```
public interface TimeSlave
    extends  TimeSynchronizable, java.rmi.Remote
{
   void setMaster(TimeSynchronizable master) throws WrongState;
   TimeSynchronizable getMaster()
}
```

(as usual, the setting of such an attribute can be done in `STOPPED` state only, hence the exception). The default value for the master is `null`, in which case the behaviour of `TimeSlave` is exactly analogous to `TimeSynchronizable`. However, if the master is not `null`, the speed value of the object, hence the transformations governing the reference point specifications, the progression speed, etc., is to be understood *relative to the clock of the master object*. This means that, for example, if the `progressPosition` operation uses the current time value to decide on the next position (see page 107), the "current time" should refer to the *master* in this case. Also, if the client changes the way the master operates (e.g., changing the units of measurement) this change will affect the behaviour of all the `TimeSlave` objects attached to the same master. On the other hand, because all slaves use the same clock, their behaviour in time is guaranteed to be properly synchronized.

Does the internal clock of the `TimeSlave` object play any role in this case? It does. Indeed, one should not forget that the internal clock of a `TimeSynchronizable` can be seen as being the "ideal" clock for the object, that is why it is internal to it in the first place. Conceptually, the state machine for this local clock is still in effect, and it is only the progression in time which is "delegated" to the master. It is therefore very important to measure whether a discrepancy exists between the clock of the master and the inherent clock of the object. For that purpose, the TimeSlave object continuously monitors the *alignment* value between the two clocks. This alignment between the two clocks can be measured as follows:

$$\left|Tick_{slave} - g(Tick_{master})\right|$$

where *g()* is a function which transforms the ticks of the master into the units of the slave, and takes into account the tick value of the master when the `reset` operation has been invoked on `TimeSlave`. A discrepancy occurs if this value exceeds certain thresholds.

The client has two means of monitoring for a discrepancy. First of all, the alignment may be inquired at any time through the operation:

```
long inquireAlignment();
```

which will return the value in the time measurement units of the `TimeSlave` object. This can help the client in implementing various monitoring policies to react against increasing discrepancies. However, this approach still requires an active inquiry; as an alternative, the PREMO event handling mechanism can also be used. Through the operation

```
public class syncHandler {
   Callback    handler;
   long        threshold;
}
void setSyncEventHandlers( syncHandler[] syncEventHandlers );
```

the client can set a series of callback references. The events are raised by the `TimeSlave` object when thresholds are exceeded by the alignment. The events themselves have the following structure tags: event name is "`OutOfSync`", event data contains one key–value pair, using "`Discrepancy`" as key and the actual alignment as a float value. (By default, no events are raised, i.e., the client has to set the threshold values and the corresponding callback references through an explicit `setSyncEventHandlers` operation request; by setting an empty array, the default is restored.) Setting appropriate event handlers the client, and indeed the application, can react to various discrepancies instantaneously, by making the `TimeSlave` object jump over certain frames, by slowing down the pace of the master, by trying to acquire additional resources, etc.

### 5.5.4 Time–Lines

The `TimeLine` object of PREMO does not add any significantly new feature to PREMO, but is a good example of how the abstractions of the various objects may be used to derive a specific, and useful object type. A `TimeLine` object is defined as a subtype of `TimeSynchronizable`, where the progression space is defined to be `Long`, and the value of `speed` is set to be of constant value 1. In other words, the abstract progression space and time are fully merged with one another in this case. This object can be used to send events at predefined moments in time to dedicated PREMO objects, and may thereby serve as a basic tool for time–based synchronization patterns:

```
public interface TimeLine
   extends  TimeSynchronizable, java.rmi.Remote
{}
```

## 5.6 Negotiation and Configuration Management

One of the main challenges of distributed multimedia applications is to cope with an extreme variety of environmental constraints. These constraints may include resource problems (e.g., processor speed, memory sizes), availability of special hardware extensions (e.g., graphics accelerators, MPEG coder and decoder hardware, audio extensions), detailed capabilities of specific services, etc. Furthermore, a distributed application may not be prepared to meet all these constraints at "compile–time", i.e., when the application is defined and installed; some of these constraints may change dynamically, and the application may be required to adapt itself to a changing environment. This adaptation may mean starting up the most appropriate object instance depending of the constraints at the time of activation, dynamically changing the behaviour of the client and/or service objects, etc. Another way of saying this is that the application is supposed to "negotiate" with its environment, and reconfigure itself in an optimal way.

This problem, as stated here, is very general and has numerous facets; PREMO cannot deal with it in its entirety. Indeed, this would, for example, require that PREMO provides a detailed specification of a number of object–oriented services which are necessary for a "smart" application management, such as metadata information about objects, rich interface repositories, global access information, etc. These specification are the subject of a series of ongoing work within, for example, OMG (so–called traders, interface repositories, etc.; see, e.g. [70,86]) and it is very probable that appropriate Java APIs will also be published in the near future. It is not the goal of PREMO to compete with these. Instead, it should rely on the general concepts underlying these specifications. In line with the charter of PREMO, i.e., to provide a reference model for "middleware" in multimedia, PREMO does not define general constraint management or negotiation algorithms either. Instead, it defines the necessary “hooks” in its object specifications, which would allow the implementation of various algorithms on the top of the PREMO objects. These hooks should also be easily usable with, for example, the newest trader services of OMG or the future Java APIs.

To answer these challenges, the fundamental approach chosen by PREMO is to build upon the general notion of properties, described in section 5.3.4.2. Properties are the basic building blocks for various configuration and negotiation mechanisms; as described already, they provide a means to specialize object instances, beyond what the type specification can do. As a general principle, the parameters governing the behaviour of objects in PREMO are described in terms of properties, rather than class attributes, if they are subject to further dynamic negotiations.

## 5.6.1 General Notions

The properties services, as defined in section 5.3.4.2, give clients the ability to attach any property to an object instance. Both the keys used by the client and the attached values are at the client’s discretion. However, for the purpose of negotiations, the specification of an object type may also *require* the existence of certain properties. This means that the object specification defines a number of property keys which are *always* present for a specific object (they are defined automatically when the object is constructed); furthermore, the specification also defines the type of values which may be assigned to a specific key, in the same way that the type of an object attribute is defined when it is defined as part of an object type. The fact that these pre–defined properties are always present makes it possible to define a negotiation mechanism based on setting and inquiring their values.

As a notational convention, we will refer to properties defined by clients, i.e., properties which are not part of the object type specification, as “private” properties. Being private means that no guarantee for the existence of these properties can be given. *This chapter deals exclusively with non–private properties.*[1)]

A negotiation based on properties requires that the client can inquire the *possible values* for a specific property key. This goes beyond what the object specification may provide; whereas the object specification may define the *type* of the property value, it cannot specify the possible *range* of values. To refer to our audio example on page 72, a property was defined to describe the possible audio formats an object may handle; however, the maximum a formal object specification may contain is that the type of the values are, as in our example, strings. Additional information may be necessary to describe that, e.g., the value of this property may contain the strings “`AIFF`”, “`AIFC`”, and only those. This additional piece of information is referred as the *capability* of the object for a key.

Things can get more complicated. Indeed, the capability describes the possible range of property values for a specific *type*. However, when an object is instantiated, the object may discover that not all admissible property values may really be used. For example,

1) Another general notational convention, adopted by PREMO, is that the names of non–private properties, as defined as part of the functional specification of objects, usually end with the character “`K`”

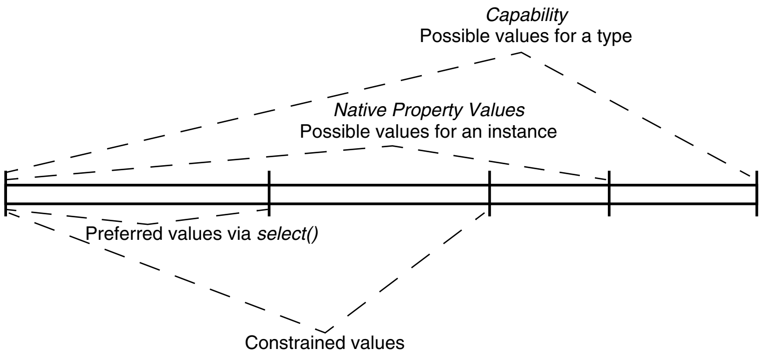

*Capability*
Possible values for a type
*Native Property Values*
Possible values for an instance
Preferred values via *select()*
Constrained values

Figure 5-18 — Type Properties, Capabilities, Constraining Properties

the object may realize that the specific graphics hardware does not allow for proper texture mapping, which means that most of the properties, which may be defined for texture mapping, may become meaningless *for this instance*. In other words, whereas the object *type* is prepared for a full generality, the object instance, which has to run in a specific environment, may not fulfill all the requirements which are defined in the object type. Consequently, there is a need for an additional piece of information, referred to as the *native property value* for a key. This is the counterpart of the capability, but for an *instance* and not for a type. The native property value constrains the values which can be set for a property.

Finally, the object may have additional knowledge about what the best values are for a specific key on a specific instance. For example, the native property value for a key describing the shading algorithm of a graphics renderer might include the values "`Phong`", "`Gourauld`", or "`Flat`". In other words, the client does have the ability to set any of the three shading methods, because all three values are permissible. However, the renderer object might be able to measure the load of the machine it is running on, as well as the resource requirements to perform a full Phong shading; based on these data, it can provide information to the client on what the optimal values are for a specific key (e.g., by stripping the "`Phong`" value, deemed to require too many resources).

Figure 5-18 gives a schematic overview of the notions involved. The object may define, as part of its type specification, the possible values for a key on the type level; this is the capability. An instance of the object type may restrict the possible values to the native property values. Both the capability and the native property values, if defined by the object specification, can be inquired by any client. If the client sets the values for this property, these values will be constrained to the permissible ones. Finally, the object may also return the set of preferred values, depending on the state of the environment.

These general notions are realized through two PREMO object types. The abstract type `PropertyInquiry`, as its name suggests, allows the client to inquire the capabilities and the native property values. The `PropertyConstraint` object, which is a subtype of `PropertyInquiry`, contains all methods which may constrain the values of properties. These objects will be described in detail in the following pages.

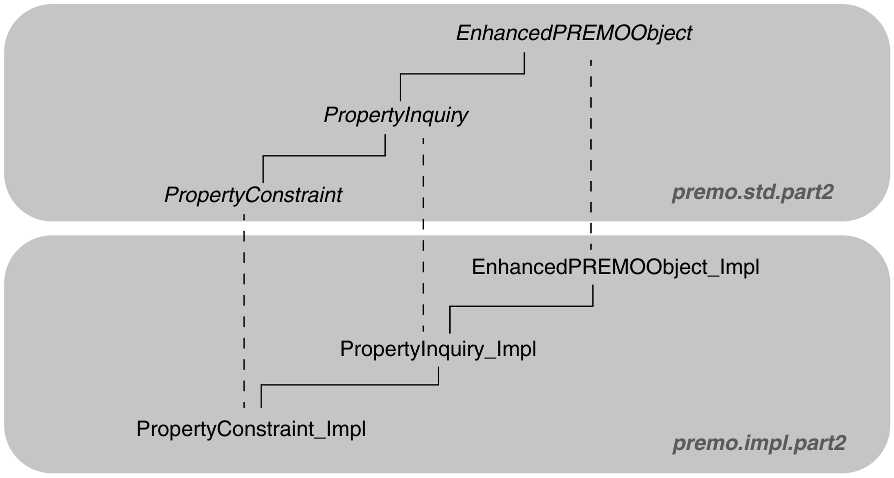


Figure 5-19 — Objects for Property-Based Negotiations

### 5.6.2 Property Inquiry Objects

`PropertyInquiry` object are subtypes of Enhanced PREMO objects so they inherit the general property management operations described in section 5.3.4.2. Furthermore, *capabilities* may also be defined for non–private properties. Although capabilities have a well specified semantics (see the previous section), from a purely formal point of view they are no different from other properties: they are defined through special, read–only properties. In other words, they are properties which provide information about other properties.[1)] To differentiate capabilities from the other properties, a naming convention is adopted: for a property name “`NameK`”, the corresponding capability can be retrieved through the property key “`NameCK`”. The value of the property for a capability (yes, it does get a bit confusing sometimes; this refers to the array for the capability, e.g., the array belonging to the property key “`NameCK`”…) is usually a list of admissible values. In some cases, the array may contain only two numerical values, referring to an admissible numerical interval. This depends on the semantics of the object.

It must be emphasized that it is *not* required that all non–private properties have a corresponding capability. For example, a property is defined as “`VendorTagK`”, denoting the vendor of a specific object (see below); although PREMO might require that all objects of a specific type should have this property defined, there is no sensible way of defining the capability to this property because the set of possible vendors is dynamic and possibly large.

1) Formally, capabilities might have been defined as special class variables, too, one per capability. However, this would have meant defining a large number of accessor methods to those class variables (using direct access to the class variables in a distributed setting is not really possible) which might have cluttered the type specifications.

Native property values are, essentially, copies of the corresponding capabilities, with possibly some values omitted. They can be accessed through a special method specified for the `PropertyInquiry` type and this is the only additional method on this interface, compared to `EnhancedPREMOObject`. Here is the full specification of the type:

```
public interface PropertyInquiry
   extends EnhancedPREMOObject, java.rmi.Remote {
      Object[] inquireNativePropertyValue(String key)
         throws InvalidKey;
}
```

Beyond the type specification, PREMO also requires the existence of some properties for the type `PropertyInquiry` objects (and hence for all its subtypes). These are[1]:

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `LocationK` | `String` | yes | Network location. |
| `VendorTagK` | `String` | yes | Implementation dependent. |
| `ReleaseK` | `String` | yes | Implementation dependent. |

The semantics of these properties are straightforward. The network location should be a string which can be used to address a node on the network, e.g., the Internet (usually the hostname, e.g., `roeiboot.cwi.nl`). The two other properties have no special semantics attached to them.

### 5.6.3 Constraining Properties

Properties may be constrained through the methods defined in the `PropertyConstraint` interface.

First of all, this interface overrides the `defineProperty` and the `addValue` methods, both inherited from the `EnhancedPREMOObject` interface:

```
void defineProperty(String key, Object[] value)
   throws ReadOnlyProperty, InvalidValue;
void addValue(String key, Object value)
   throws ReadOnlyProperty, InvalidValue;
```

Compared to the “original” versions, both methods have one more exception that they can throw, namely `InvalidValue`. This happens if the value or values are outside the range of the native property value of the object (for the specific key). The data in the exception returns the unacceptable values, and the properties are not changed if an exception is raised.

The type defines an additional property setting operation which can be used to “collect” a series of properties and set them all at once:

```
PropertyPair[] constrain(PropertyPair[] constraints)
   throws InvalidKey, InvalidValue;
```

[1] Formally, a property is defined as an array of values, see page 71. The “Type” column refers to the type of the objects appearing in this array.

(The PropertyPair non–object type has been defined on page 72.) The method attempts to set all the properties as defined in its argument; however, for all property keys, it also checks the values, which should be within the range defined by the native property values. In other words, it takes the intersection of the property values in the argument and the native property value for the same key. The InvalidValue exception is only raised if one of the values is of an inappropriate *type*. Instead of raising an exception if a value is not permitted (e.g., not listed as an acceptable native property value), the value will be simply ignored. On the other hand, the operation returns the new set of values automatically, i.e., the client can check the result of the constraining operation. This operation is particularly useful in a distributed environment where the number of operation calls should be kept low; using constrain the client can set the properties in a batch.

The select operation gives a much more active role to the PropertyConstraint object. Formally, it is similar to the constrain operation above:

```
PropertyPair[] select(PropertyPair[] constraints)
   throws InvalidKey, InvalidValue;
```

However, this operation not only calculates the intersection of the values with the native property values; it may also apply some further selection, thereby restricting the accepted property values. Of course, the restriction depends on the exact type of the object performing this operation; obviously, by default, the effect of this method is identical to constrain.

### 5.6.4 Dynamic Change of Properties

Constraining properties to specific values may also be related to the *state* of the object. The problem is that the distinction between read–only and general properties does not account for the fact that, in some cases, the lifetime of an object may clearly be divided into a "configuration" phase and a real "working" phase; depending on the phase the object is in, a property may have to be static, i.e., read–only, or changeable. For example, properties may control the sample size and the sample rate of some data, such as audio. These properties may be set in a configuration phase; however, once the media stream flows, neither the client nor the object itself should change these values, i.e., the properties should become (temporarily) read–only.

Changeable properties may again fall into two categories: mutable or dynamic. Mutable properties are such that no *client* should be able to change these values when the media stream flows, although the object itself may change them. Such values might, for example, change as a result of decoding a protocol found within the media stream which defines the mutable property value (e.g., the quantization matrix of an MPEG flow). Finally, there may be properties which may be changed at any time. These are referred to as dynamic properties.

To manage these situations, the PropertyConstraint object should be viewed as a tiny finite state machine which has two states: bound and unbound. There are two methods to change the state of the object:

```
void bind() throws InvalidValue, java.rmi.RemoteException;
void unbind() throws java.rmi.RemoteException;
```

(the reader should disregard the `InvalidValue` exception for now, we will come back to it in the next section). When in the bound state, properties which are not defined to be fully dynamic become read–only. The corresponding exceptions are raised if the client attempts to change these values through, e.g., the `addValue` method. Once the state of the object becomes unbound again, the property values may be changed.

How does the client know which are the dynamic and the mutable properties, or none of the two? Of course, through additional read–only properties again… PREMO defines the following read–only properties for `PropertyConstraint`:

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `MutablePropertyK` | `String` | yes | List of mutable property keys. |
| `DynamicPropertyK` | `String` | yes | List of dynamic property keys. |

i.e., these properties may be inquired to find out which *other* properties are mutable or dynamic. In other words, if the state of the object is bound, only the properties, whose keys are listed in the property value for `DynamicPropertyK`, can be changed; all other properties (the non–private ones, that is) become temporarily read–only.

These properties are the first examples where capabilities are also defined:

| Key | Type | Description |
|---|---|---|
| `MutablePropertyCK` | `String` | Type dependent list of mutable properties |
| `DynamicPropertyCK` | `String` | Type dependent list of dynamic properties. |

Of course, these capabilities have a limited interest only, because the corresponding keys are read–only, i.e., the property constraining mechanism cannot be applied. However, because a capability is defined, a native property value is also available, and this may be of real interest. Indeed, if the native property value provides a subset of, say, what is listed in the `DynamicPropertyCK` array, this means that the object instance imposes a further restriction on the changing of certain properties, compared to what the object type allows.

At construction time (and only then, of course!) each subtype of the type `Property-Constraint` may add its own values to these capability arrays, i.e., if a key is defined to be, e.g., dynamic, it remains dynamic in subtypes, too.

### 5.6.5 Interaction among Properties

The mechanisms described in the preceding section are missing a feature. All property constraining mechanism hitherto described refer to *one* property key only. In other words, how one property value is set for a specific key is independent of the values set for another key.

However, in reality, mutual dependencies also occur. Referring to an audio object again, it may have two properties, say, sample size and sample rate. In our example, the sample size can be of 8 bits or 16 bits, while sample rate can be 8 KHz or 40 KHz. If all combinations of properties are possible, then the possible options are

| | **Sr=8KHz** | **Sr=40KHz** |
|---|---|---|
| **Ss=8bit** | Sz=8bit, Sr=8KHz | Sz=8bit, Sr=40KHz |
| **Ss=16bit** | Ss=16bit, Sr=8KHz | Ss=16bit, Sr=40KHz |

The complication is that, in practice, media objects abstract real media devices. These media devices often allow only restricted combinations of property values for a specific instance. The audio device, for example, could support the following combinations only:

| | **Sr=8KHz** | **Sr=40KHz** |
|---|---|---|
| **Ss=8bit** | Sz=8bit, Sr=8KHz | |
| **Ss=16bit** | | Ss=16bit, Sr=40KHz |

in other words, only certain combinations of property values are acceptable.

To remedy the problem, yet another pair of property and capability value is defined for `PropertyConstraint`, which describes the permissible combinations of properties. The property is as follows:

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `ValueSpaceNameK` | `PropertyPair[]` | yes | List of mutual property dependencies. |

What does this property describe? First of all, the property value is an array of the "Type" in the table, i.e., it is an array of an array of property pairs. That means that each element of the array describes a sequence of possible key–value combinations, which refer to non–private keys of the object. When checking the correctness of the property settings, one of the elements of this outer array should describe a combination which fits with the current values.

What does it mean "fitting with the current values"? All non–private properties of the objects should be considered, and these values should be compared to the values listed in the `ValueSpaceNameK` array element. If the current values are all subsets of the required values, than the values are accepted. If this is true for all elements appearing in the `ValueSpaceNameK` array element, than the overall property settings for the object instance are accepted. (There are some boundary conditions: if a key does not appear in a `ValueSpaceNameK` array element, this means that the combination does not impose any further constraints on this property, i.e., all values are accepted; finally, if the full `ValueSpaceNameK` array is empty, than no mutual dependency is imposed upon the properties of the object at all.) A capability, `ValueSpaceNameCK`, is also defined for the object with the obvious meaning.

A question remains: when is this mutual dependency check done? This is done by the `bind` operation. Recall that this operation (see page 115) changes the state of the object from unbound to bound; essentially, this operation closes the "configuration phase" in the lifetime of the object. However, before changing the state of the object, the operation performs the check on the property values of the object, using the `ValueSpaceNameK` property, as described above. If a problem is found, the state of the object does *not* change, and an `InvalidValue` exception is raised. The exception instance will contain an array of property pair classes, listing the key and value pairs which were not accepted. Using this information, the caller can check where things went wrong in setting the properties of the object.

### 5.6.6 Some Conclusions on the Negotiation Facilities

The mechanism which has been described in this chapter by no means offers a full solution for all possible constraint problems. For example, PREMO does not include explicit management for general constraints, such as geometric constraints, although this might be a very important feature in practice. This decision was not easily made, and was the result of long and sometimes passionate discussions within the PREMO team. There is indeed a classic tension between the general requirements of constraint management and the essence of object–orientedness. Whereas the latter advocates information hiding, the former requires a complete knowledge of all the attributes related to an object (see, for example [27]). It was recognised that there is no widely accepted object model which would solve this problem in a satisfactory manner and in general terms. Because PREMO is an international standard, i.e., a platform for general consensus, the development team finally decided not to include a fully general mechanism for constraint management.

However, the property constraining mechanism is a very powerful tool, and can be used to implement a large number of algorithms. It uses concepts which are very familiar in the distributed object world, like the OMG property services, which makes it easy to implement and match with external facilities. As we will see in other chapters of this book, its level of abstraction is quite appropriate for the kind of negotiations PREMO tries to cover, without requiring very complex concepts — this is, in fact, its major strength.

## 5.7 Creation of Service Objects

All previous sections were silent on one important issue: how are PREMO objects created?

This seems to be a simple problem at first glance. Practically all object–oriented environments, including Java, have language features to create new instances of objects. In Java, for example, one uses the `new` statement which creates a new and properly initialized instance of a class. However, when an application is embedded into a distributed environment, things are much less obvious. An object may have to be created on another node on a network (i.e., within another instance of a Java Virtual Machine, in the case of Java) which cannot be done directly with `new`. The newly created object must be "exported" somehow as a server object on the network, etc. Furthermore, in most cases, the caller (i.e., the object which initiates object creation) receives a reference to "stub" object rather than to the real object itself. This must also be handled.

Obviously, the intimate details of object creation belong to the somewhat grey area which separates the pure PREMO world from its implementation environment, and PREMO cannot control all aspects of this process. Just as in the case of the general property and negotiation management facilities (see page 110), the approach of PREMO is to specify only a few, rather abstract objects which describe what is necessary in terms of PREMO, and leave all the details for the implementors of these objects. In the case of object creation, two such objects are defined: generic factories and factory finders.

### 5.7.1 Generic Factory Objects

The purpose of the generic factory object is to provide a wrapper around the object creation facilities, but taking a list of property requirements into consideration, too, when creating an object. These properties describe the required characteristics of the object to be created.

The interface of the object is relatively simple:

```
public interface GenericFactory
   extends PropertyInquiry, java.rmi.Remote
{
   PropertyInquiry createObject( Class          objectType,
                                 PropertyPair[] constraints,
                                 Object         initValue)
      throws   InvalidCapabilities, CannotMeetCapabilities,
               InvalidType, IncorrectInit;
}
```

however, this apparent simplicity hides a very complex operation. The semantics of the call is as follows.

The goal is to create an instance of the type `objectType`, and to initialize this object instance with the value `initValue` (through the `initialize` operation, see page 64). This object creation may be remote, i.e., the object instance itself may run on a remote node, and only a reference to this remote object (or its stub) is to be returned. In other words, the factory implementation should hide the peculiarities of remote object creation.

Furthermore, a single factory instance may have the ability to instantiate various embodiments of the same object type. For example, the factory may have access to all computers within an internal network and it could therefore create an instance on any node of this network. Which node should it choose? How should it control the choice?

This is where the `constraints` argument comes into play. This argument is an array of property keys and corresponding values; the goal is to control the properties of the object to be created. To be more precise, the *native property values* of the new object instance should be a *superset* of the values appearing in the `constraint` array. Recall (see section 5.6.2) that the native property values tell us the possible values for a property an object instance may have. The role of the `constraint` argument is therefore to define some kind of a minimal capacity of this object. Of course, the factory may not be able to fulfil all these requirements; various exceptions are defined to designate failure. These exceptions are quite self–evident.

A full–blown implementation of a generic factory object may be very complicated. It relies on a complex infrastructure governing a distributed object environment. Interface repositories should be available where descriptions of the various possible object instantiations are made available to factories, which can then choose the optimal instances; access to remote instantiation procedures should be provided to create an object on a remote node (by setting the `LocationK` property within the `constraint` argument, the caller of the `createObject` method can control where the new instance should run!), etc. Fortunately, such infrastructures are emerging, both in the OMG world, as well as within Java[1)], which makes the implementation of powerful generic factories feasible.

The attentive reader may have realized that the return type of `createObject` is `PropertyInquiry`, and *not* `EnhancedPREMOObject`. Indeed, only these objects have a native property value defined. This also means that a majority of the objects defined in Part 2 of PREMO *cannot* be instantiated through a generic factory. They have to be instantiated locally, albeit making them available to the full distributed setting. This may seem as a restriction at first glance. However, as it will become clear in the chapters to come, all “big” objects, abstracting virtual multimedia devices, renderers, etc., are, in fact, `PropertyInquiry` objects. They may have internal instances of other objects, such as event handlers, which may have to be exported, but they usually control the creation of these simpler entities. As a consequence, factories are not really needed for the creation of the objects which are not `PropertyInquiry` objects, too.[2)]

[1)] At the time of writing, the interface repository facilities of Java are quite simple, but we can be sure that by the time this book appears on the bookshelves, much better facilities will be available.

[2)] Strictly speaking, this is not always true. Exporting an object through RMI in Java requires some additional calls beyond the simple construction of the objects. Of course, these statements could be added to the constructor of all the objects, thereby hiding the problem. However, experience shows that access to objects which are exported as RMI servers is somewhat slower than accessing them directly, *especially if accessed within the same JVM*. As a consequence, optimization may require to separate object creation from their export through RMI, and this separation leads to separate facilities for object creation. This is, however, and optimization issue which is not, and should not be, addressed by PREMO.

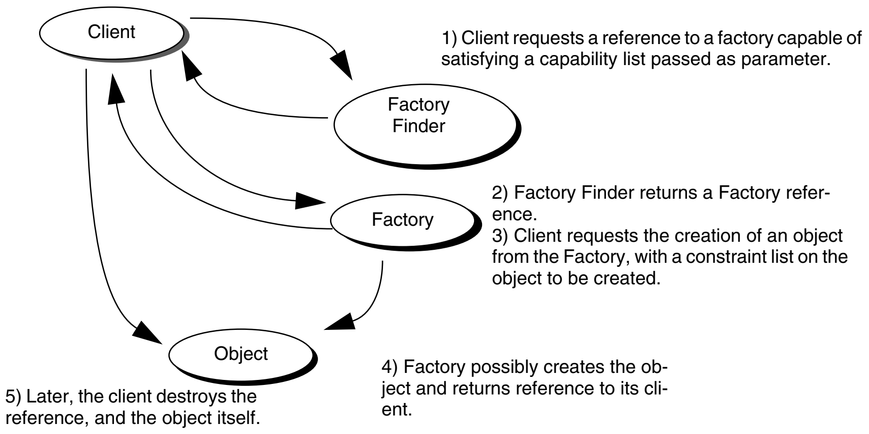

Client
Factory Finder
Factory
Object
1) Client requests a reference to a factory capable of satisfying a capability list passed as parameter.
2) Factory Finder returns a Factory reference.
3) Client requests the creation of an object from the Factory, with a constraint list on the object to be created.
4) Factory possibly creates the object and returns reference to its client.
5) Later, the client destroys the reference, and the object itself.

Figure 5-20 — Use of Factory Finders and Factories

### 5.7.2 Factory Finders

Of course, to use a factory, the caller must have access to it. In other words, a reference to a factory object has to be located. This is done by a separate object, called the factory finder object, whose specification is as follows:

```
public interface FactoryFinder
   extends EnhancedPREMOObject, java.rmi.Remote
{
   GenericFactory[] findFactories(
         Class          objectType,
         PropertyPair[] objectConstraints,
         PropertyPair[] factoryConstraints )
      throws InvalidCapabilities, CannotMeetCapabilities,
            InvalidType;
}
```

The goal is to locate an set of factories, which have the following characteristics:

- they can all create an object type of `objectType` (which must be a subtype of `PropertyInqiry`);
- the capabilities of the objects of type `objectType` should be a superset of the property values described in `objectConstraints` argument; and
- the capabilities of the factory objects themselves should be a superset of the properties described in `factoryConstraints` argument.

All three arguments may be `null`, meaning that the corresponding constraints do not apply. For example, the value of `objectType` may be `null`, governing the `FactoryFinder` object to locate a set of factories with general capabilities described by the argument `factoryConstraints`.

The semantics is a bit similar to the behaviour of the generic factory itself. The idea is to locate a set of factories which will be able to create certain types of objects. The constraints used for this purpose are simply more specifically tailored to the need of factory access. Here again, just like in the case of the factory objects themselves, the operation may not find the appropriate factories, either because it does not have necessary information available, or because appropriate factories are not accessible. Exceptions are raised in this case with an obvious meaning.

Of course, the "recursion" could continue; in order to find a factory finder object, a finder of factory finders should be defined, etc., and this could go on ad infinitum. However, PREMO stops at this point. It is implementation dependent how an application can access a factory finder.

### 5.7.3 Use of Factories and Factory Finders

We emphasized on page 120 that the full–blown implementation of a factory (as well as of a factory finder, as a matter of fact) can be very complex, and it relies heavily on the facilities provided by the implementation environment of a specific PREMO implementation. In what follows, only a very simple scheme is shown how factories and factory finders may operate in practice. We emphasized in Chapter 4 that this book relies on a prototypical implementation only, which waives a number of issues and complexities, concentrating on the main points only. However, even such a simple scheme may help in understanding how these object may cooperate.

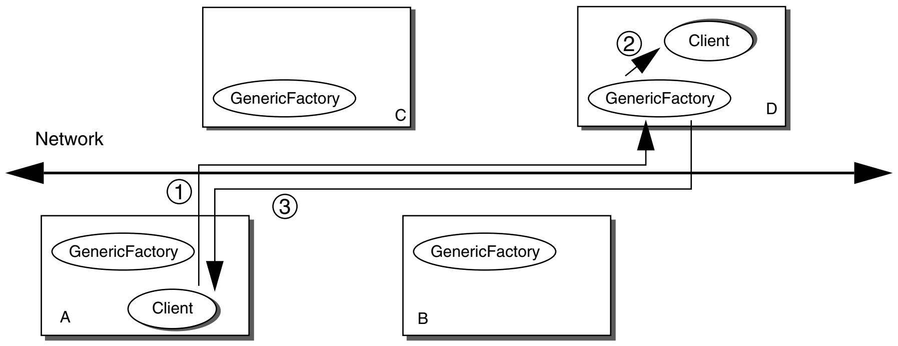


Figure 5-21 — Simple Use of Factories

Figure 5-20 gives a schematic view of how a client, a factory finder, and a factory cooperate in the life cycle of a PREMO object instance. In fact, this scheme is still independent of any implementation, and reflects the general notions described earlier. Figure 5-21 shows how factories operate in our practice. Each box represents a separate Java Virtual Machine, running PREMO, and cooperating through Java RMI. In this example there is only one JVM running on a specific network location (i.e., one JVM per machine) which simplify the identification of a JVM. Each virtual machine runs one instance of a generic factory, and only one. This instance is also exported to RMI, i.e., it operates as a service over the network. The restriction in our case is that a factory instance can create objects within its own JVM only.

If an object, running in JVM marked "A" on the figure, wants to instantiate another object within the JVM marked "B", it must refer to the factory objects assigned to JVM "B", and invoke its `createObject` operation (this invocation goes through the Java RMI facilities). The factory object of JVM "B" will create an object instance, initialize it, export its reference to RMI to make it a service object, and return the reference to the object to the caller. To be somewhat more precise, the request to the factory object refers to an object type *as defined in PREMO*, i.e., it refers to an interface defined in one of the `premo.std.partX` packages. In view of our "dual" scheme of object interfaces and their implementation (see section 4.2.1), the factory will instantiate the "`_Impl`" counterpart of the interface, i.e., its implementation, and a stub reference to this implementation instance will be returned to the caller.

Each node also runs one and only one instance of a factory finder object (not shown on the figure). The role of the factory finder objects is to locate the various Java JVMs, using their network location name. This network location name can be requested by the user of the factory finder object by filling in the "`LocationK`" property in the `factoryConstraint` argument of the `findFactories` call. Because in our example there is only one generic factory running per network location, the hostname is enough to extrapolate the name of the generic factory object, and locate it through the standard Java RMI facilities (the details are not really of importance here). Slightly more complex facilities can also be easily implemented (e.g., underspecifying the host domain name in the location, allowing the localization of a range of factory objects, etc.).

Using these facilities, here is how a generic factory, running on the host called "`hydra1.cwi.nl`", can be located (the `PREMORuntime` class, used in the example, is an object which collects some general, static variables and methods which are necessary to start–up and run PREMO applications):

```
FactoryFinder fFinder   = PREMORuntime.localFactoryFinder;
PropertyPairlocs[]      = new PropertyPair[1];
Object[] vals           = new Object[] { "hydra1.cwi.nl" };
PropertyPairlocation    = new PropertyPair("LocationsK",vals);
locs[0]                 = location;
GenericFactory CWI      = (fFinder.findFactories(null,null,locs))[0];
```

of course, this is a bit long, because all data structures had to be created from scratch. If, as a next step, the factory on the node "`minster.york.ac.uk`" is to be located, this is simply done by:

```
location.vals[0]        = "minster.york.ac.uk";
GenericFactory York     = (fFinder.findFactories(null,null,locs))[0];
```

Note that no further constraints have been imposed on the factories this time, not even for the type of objects they can create.

Using the factories retrieved above, new object instances are created easily (again, the property constraint features are not used):

```
AudioDevice vDev = (AudioDevice) CWI.createObject(audioClass,null);
Renderer ren = (Renderer) York.createObject(rendererClass,null);
```

which will create a virtual device and a renderer object, one on a machine on the CWI domain in the Netherlands, the other on a machine at York, in the UK (`audioClass` and `rendererClass` are supposed to be `Class` objects referring to the relevant PREMO interfaces). Provided the network throughput is fast enough (which is never the case!) the two authors of this book can then cooperate through these objects for the purpose of multimedia rendering…

# Chapter 6

# Multimedia Systems Services Component

## 6.1 Introduction

"Multimedia Systems Services" (MSS) was the name given by the Interactive Multimedia Association (IMA)[1)] to a model for distributed multimedia applications. This model was specified by a working party of IMA which grouped representatives of various multimedia technology and content providers as well as workstation manufacturers. Later, in 1995, this specification was submitted to ISO to be included in the PREMO standard. After some major editing work, which significantly influenced other parts of PREMO, MSS has become a core component of PREMO (as Part 3 of the document), with its original name retained.

The original goal of the MSS was to provide "an infrastructure for building multimedia computing platforms that would support interactive multimedia applications dealing with synchronized, time–based media in a heterogeneous distributed environment". Operation in a distributed environment was considered to be very important, in line with the significant trends in the computer industry towards client/server and collaborative computing. While achieving the original goals, the model put forward by IMA has proven to be very powerful, and potentially applicable as a conceptual model for multimedia processing in general, not only for distribution. This is the role it now plays as part of the PREMO standard.

The conceptual model of MSS is based on a dataflow network of devices, the so–called *virtual devices*, each of which is an autonomous processing unit (see Figure 6-1). The nature of the processing (capture, encoding, filtering, display, etc.) varies according to the specific device object (and is implemented through subtyping). The directed links among the virtual devices are the *media streams*, which serve as a way to send media data from one node to the other. Each virtual device has a number of *ports*, which can be either input or output, and which are used to convey the content of the media streams to and from virtual devices. The number of ports, as well as their "direction", is device specific. A device may have no input port at all (for example, by producing an animation sequence on its output port, based only on some internal data), or have no output port (for example, a graphics display engine, which receives data to display, but does

1) The Interactive Multimedia Association (IMA) is a large, international trade association of multimedia technology providers, multimedia content providers, and users. For further details, see their web page: `http://www.ima.org`.

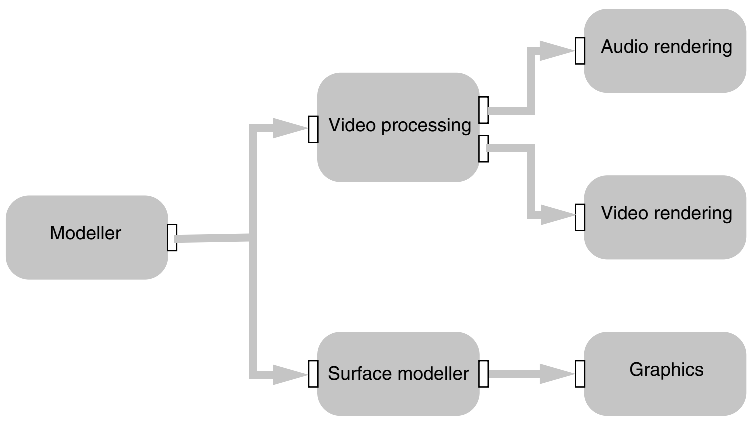


Figure 6-1 — Dataflow Network of Virtual Devices

not forward media data to other MSS devices). By “plugging” various virtual devices together, complex processing networks can be created with extremely rich functionalities.

It must be emphasized that, although the word “distribution” has been used, MSS does *not* require that the virtual devices be necessarily distributed in the traditional sense, i.e., over a physical network. If we refer to *parallelism* as the application of more than one processor to carry out a solution of a problem (of which distribution is a special case), and to *concurrency* as carrying out a set of activities which overlap in time, then the MSS model refers to concurrency rather than parallelism. Although virtual devices may indeed run on different machines on a wide–area or a local–area network, they may also run as concurrent processes within a single machine, or as threads within the same process. The model does not specify any of these choices, and it is up to the implementation of MSS whether they provide all these facilities or only some.

The use of a dataflow model for the description and control of multimedia systems was not invented by MSS. The approach taken in the multimedia framework described in [65] (see also [34]), or in various packages for scientific visualization, such as AVS, is very similar to that of MSS. This model is also very natural when dealing with true distribution, where any sort of synchronous control of concurrency essentially breaks down.

MSS defines the so-called `VirtualDevice` object type, shown in Figure 6-2. This consists of a collection of input and output ports, through which media data can flow in and out. The real processing of a device (the “processing element” in the figure) is not specified within PREMO as an object type, nor is the means by which this element communicates with its resources specified. The goal of MSS is simply to ensure that, through a well specified interface, virtual devices can cooperate properly. Devices for specific media are to be defined by subtypes of virtual devices. To ensure a clean co–operation, various objects are associated to each port that together characterize the nature of the communication that may take place via that port. This characterization is

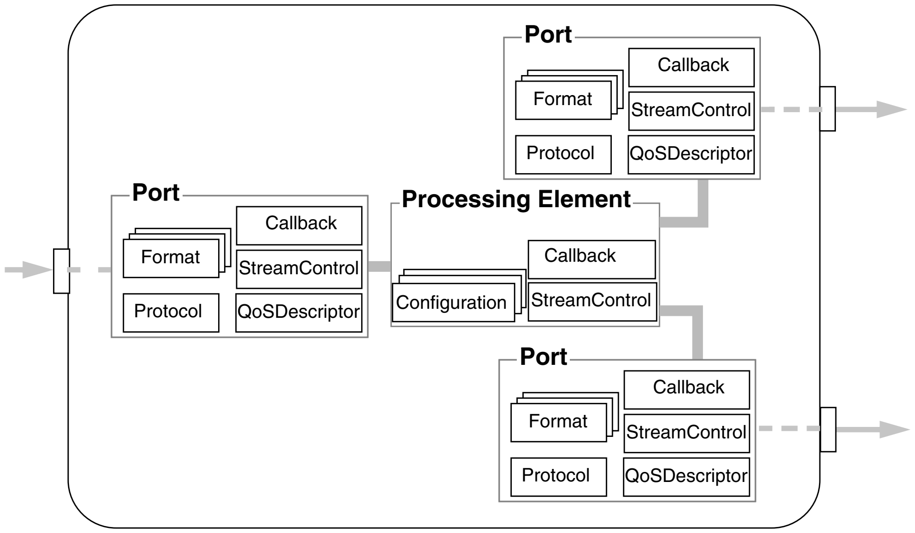


Figure 6-2 — A Virtual Device

done through object properties. These objects are instances of the types `Format`, `Protocol`, `QoSDescriptor`, and `StreamControl` (all these object types will be described in more detail later in this chapter).

One of the fundamental features of MSS is that clients can *configure* a dataflow network. This configuration is done in a negotiation phase, when a client retrieves information about the virtual devices it has at its disposal, and tries to "plug" a media stream into a matching pair of ports. This mechanism is based on the general negotiation and configuration management tools provided by the foundation component of PREMO, see section 5.6 (page 110). Using these tools, the client can make sure that, for example, the media format produced on a port of a virtual device can be understood and decoded by the receiving port. MSS defines general objects (so called *virtual connection* objects), which give finer control over this mechanism.

MSS does not define the detailed *behaviour* of media streams in terms of networking or other forms of communication. It only defines the ways through which a client can control and possibly synchronize the flow of media data on either end of the stream (there is no such *object* as a "`Stream`" in PREMO). From the point of view of MSS, streams are abstract communication channels among devices. The only requirement is that they must provide an order–preserving and reliable communication. How the communication is actually realized will depend on the kind of environment in which the PREMO system is running.[1)]

[1)] Unfortunately, there is an conflict of terminology here: "streams" in the PREMO sense and "streams" in the Java sense are not identical although they have similarities. To make things even more confusing, Java streams may indeed be used to implement PREMO streams…

MSS does not define what the *content* of the media data is either. The goal of the MSS is to concentrate on the facilities for configuration, negotiation, etc.. The details of the data formats which constitute the media data are left to the specific devices. This is one point where various other multimedia specifications and standards meet with PREMO. As an example, while the ISO MPEG specification describes the details of a video format (in the terminology of PREMO, it describes the media data details flowing through a media stream), PREMO concentrates on how an MPEG coder/decoder (which can be abstracted by a virtual device) can be used together with other media processing entities.[1)]

MSS also provides tools to create hierarchies of dataflow networks. A special subclass of virtual device, called a *logical device*, may contain a full dataflow network of virtual devices, which is invisible to the external observer. Ports of such logical device are merely transition points toward ports of “internal” devices. MSS also defines other object types which help to manage, create, or control dataflow networks. In a truly object–oriented fashion, MSS defines the *virtual resource* object type as a common supertype for virtual devices, logical devices, and various controlling objects. These are the objects which are typically created (through an object factory mechanism) and accessed directly by the client.

All in all, MSS is a real *middleware*, in the sense that it provides a standard model for various entities to cooperate, but it does not specify the detailed behaviour of these entities. This also means that there is a delicate balance in the specification of each MSS object; it should not be too detailed, otherwise it would restrict the family of cooperating entities, but it should not be too vague, either. This strive for balance underpins the specification of each object described in this chapter.

Figure 6-2 also shows that virtual devices, and virtual resources in general, are like puzzles, insofar that they aggregate a number of MSS object into one logical entity. In what follows, the basic constituents of virtual resources will be presented in more detail; these are the so–called configuration objects and the stream control objects. These objects are then used to build up the virtual resources, which will be presented in a separate sub–chapter.

## 6.2 Configuration Objects

The family of MSS object types, categorized as *configuration* objects, are used as information depositories for other objects (see Figure 6-3). The role of configuration objects is to act as placeholders for the necessary parameters, i.e., for *information*, which allow other objects to function properly. These parameters may describe, for example, media coding (AIFF, MPEG, JPEG, CGM, etc.), communication protocol types (TCP, NETBIOS, ATM), and the like. One could view them as sorts of attributes whose setting and retrieval follow a more complicated pattern than simply read and write values.

[1)] It must be noted, however, that Part 4 of PREMO (i.e, the Modelling, Rendering, and Interaction component) goes beyond this, and it does define, up to a certain extent, the content of media flow with a particular application area in mind. However, it is perfectly feasible to build up valid applications using the facilities provided by the MSS only. It is therefore important to keep this distinction in mind.

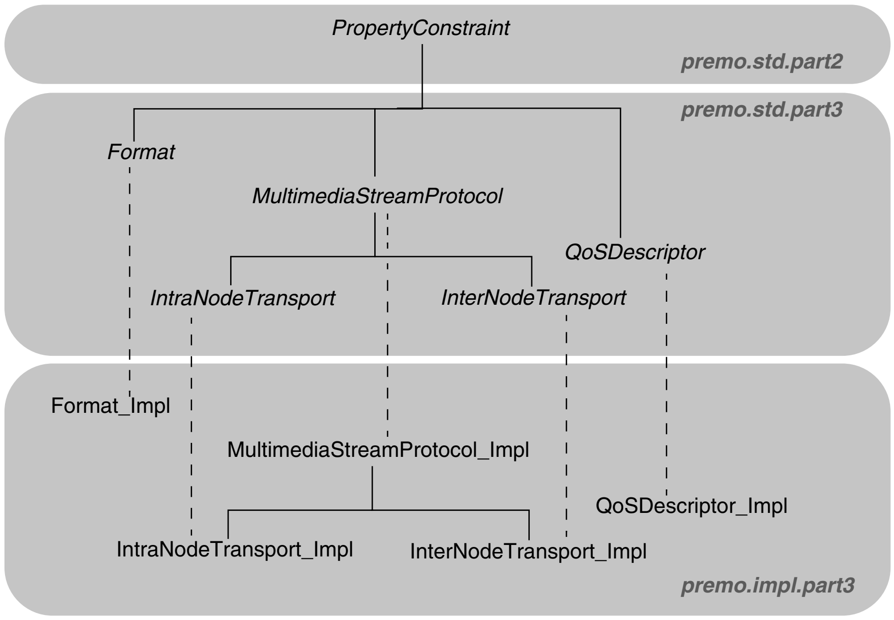


Figure 6-3 — Configuration Objects

This "attribute–like" behaviour is realized through the subtyping relationship which binds configuration objects to the rest of the PREMO hierarchy. All configuration objects are subtypes of `PropertyConstraint` objects (see section 5.6 on page 110), i.e, the various values they provide to clients are stored and manipulated through the property constraining mechanisms which characterize the `PropertyConstraint` objects. Typically, configurable MSS objects, such as virtual resources, contain several instances of configuration objects, whose references can be accessed by external clients. These clients can then set the required values for the configuration objects through the `constrain` and `select` operations.

The use of configuration objects is the basic mechanism for configurability in PREMO. Clients may inquire the key–value pairs associated with these objects and, through the property constraining and selection mechanism, may restrict these values. Configuration objects do not define additional methods. Various subtypes of configuration objects differ from one another only through the property keys, capabilities, and native property values they define. This might give the impression that configuration objects are simple. However, one should not forget that the behaviour inherited from `PropertyConstraint` objects, especially the `constrain` and `select` operations (see page 114), represents a significant level of complexity when it comes to specific configuration objects. Subtypes of configuration objects may be defined with a constraint algorithm, which is made available through the `select` operation. These algorithms may take into consideration the full set of configuration objects associated with the same device, or with a set of cooperating devices. However, details of these algorithms are not specified by PREMO (beyond the inherited behaviour of the `select` operation).

As `PropertyConstraint` objects, configuration objects also act as tiny two–state finite state machines. Objects can be "bound" through the `bind` operation (see page 115), which means that even writable properties may become temporarily fixed. Typically, configuration objects are automatically bound when the media data begins to flow. This will be more fully explained later. What this means, however, is that the management of the information through configuration objects is typically done in a separate, "negotiation" phase, prior to any media flow. It is only when the negotiations are over that the media data is allowed to enter or leave a virtual device, or a set of virtual devices.

Configuration object instances belong to a virtual resource instance just like attributes belong to object instances. Clients are not supposed to *create* configuration objects. Instead, these objects are created by the virtual resources themselves, and clients can retrieve the configuration objects related to a specific virtual resource instance. This view is reinforced by the concept of *semantic names*. Semantic names are simply strings and are used to identify configuration objects used by a virtual resource. To access configuration objects, virtual resources (e.g., all virtual devices) have a separate property which lists the semantic names of the configuration objects they define for themselves. Using the standard property inquiry operations, clients can retrieve these names. Virtual resources also have an operation called `resolve`:

```
PropertyConstraint resolve(String semanticName)
```

which can be used by clients to get the object reference of a specific configuration object instance.

As an example, the semantic name for a specific device might include, for example, the strings "MPEG" or "ATM". This means that the device can operate with these video formats. The call

```
PropertyConstraint protocol = dev.resolve("ATM")
```

will return a reference to a configuration object, representing an ATM protocol description, *but characteristic to the device which "owns" it*. Details of all these operations will be given later.[1)]

Configuration objects of PREMO fall into three categories:

1. Format objects
2. Transport and Media Stream Protocol objects
3. Quality of Service objects

In what follows, the use of each of these categories will be presented.

[1)] There is no way in Java, or in most of the object oriented environments for that matter, to restrict which other object can create a specific object instance (defining a constructor to be `protected` or to have a package visibility only may be much too restrictive in practice). Consequently, this restriction on configuration object creation cannot be reinforced by the environment. It can only be a convention when using PREMO.

### 6.2.1 Format Objects

Format objects are the most genuinely multimedia configuration objects; their role is to give information on the media formats (that is, the organization of the bitstream) of a particular device, or at a particular port of a device (see Figure 6-2 on page 127). The characteristics of a specific media format are described in the form of object properties.

The formal specification of a `Format` object is simple:

```
public interface Format
   extends premo.std.part2.PropertyConstraint, java.rmi.Remote {}
```

which corresponds to the fact that configuration objects do not define any more operations beyond those which are inherited from `PropertyConstraint`. Furthermore, PREMO defines a read–only property for a `Format` object, namely:

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `NameK` | `String` | yes | Semantic name of the object. |

An example should help in understanding the structure and use of format objects (note that, in the example below, we anticipate methods for virtual devices which will be defined more formally later). In this example, we take a very simple virtual device, which has only one input port. The device can receive a digital image on this port and can display the image on a screen. Because there is a large number of image formats used in practice, the device is prepared to receive image data in various formats, such as GIF, JPEG, PNG, TIFF, etc.

For each of these formats, a separate `Format` object type is defined by the implementation of the device, e.g.:

```
public interface JPEGFormat extends Format, java.rmi.Remote { }
```

with, of course, the corresponding implementation classes. Each of these format objects should have an appropriate semantic name, say, the strings "`GIF"`, "`JPEG"`, "`PNG"`, "`TIFF"`, etc.

The client can find out which formats the device can use through the call:

```
ConfInfo[] confs = dev.getProperty("ConfigurationNamesK");
```

The structure `ConfInfo` (specified on page 141) contains the available semantic names. In other words, the strings "`GIF"`, "`JPEG"`, etc., are contained in the elements of the `confs` array. This means that the client can find out which image formats the device is able to handle. Based on some environmental constraints, the client may choose, for example, "`GIF"` to be its preferred image format. This can be expressed by setting the so–called port configuration of the device's port, which essentially means to set the "`GIF"` format instance for the port (details of this step will be given later).

The `GIFFormat` object may have a number of properties which characterize GIF images. For example, it may have a property key "`GifVersionK`", with possible values "`87a`", "`89a`", differentiating between the two versions of the GIF specifications which are widely in use. It may have boolean property values for the key "`Transparency`", denoting whether the device instance can handle transparency or not, properties keys for normal vs. interlaced images, colour characteristics, etc. Some of these properties have mutual dependencies because not all combinations are allowed (e.g., transparency is only supported by GIF version 89a). The permissible combinations are described through the "`ValueSpaceNameK`" property (see page 117). In short, the properties allow for a very rich characterization of the format.

If necessary, the client can also obtain the object reference for the format object:

```
GIFFormat gifFormat = (GIFFormat) dev.resolve("GIF");
```

and use this reference to access these property values and, if necessary, set them. How these properties are used depends on the client. In some cases, these properties are used for information only. In other cases, these properties are also set through the property constraining mechanism. In our example, the client may want to connect the image displayer device to another device which produces the images. If the GIF version accepted by the image displayer is "`87a`" only, and this information is conveyed through the "`GifVersionK`" property, then the client must *set* this value on the GIF format object belonging to the image producer, otherwise a discrepancy will occur. This type of configuration setting must be done for all different format objects in a device network. Obviously, managing and matching all the format characteristics may be very complicated but, unfortunately, this is the reality in multimedia processing.

The core PREMO document does not define format objects for specific media, like the `JPEGFormat`, `GIFFormat` formats above. Only the general mechanism is defined, and other expert groups, standardization bodies, or simply PREMO implementations are supposed to define the details. However, an informative annex of PREMO does contain an overview for some typical format objects, which can be used as a starting point for a more precise specification. It is not the purpose of this book to describe all possible details. The interested reader should consult the ISO PREMO document for these.

### 6.2.2 Transport and Media Stream Protocol Objects

It has been emphasized before that virtual devices may be distributed over a network, or over several processes within the same machine (in terms of Java, over several instances of Java virtual machines). Connecting two devices through a network requires some careful considerations, involving the exact communication protocol to choose, how to connect the devices, etc. PREMO defines a distinct type of virtual resource, whose only purpose is to connect devices through, possibly, network connections. These objects, called *virtual connection* objects, will be described in more detail later in this chapter (see section 6.6).

The purpose of the Transport and Media Stream Protocol (MSP) configuration objects is to provide information on which protocol is used to convey media data among processing nodes. It is not the role of MSS to give a detailed specification of the various possible communication protocols, only references to existing protocol specifications are made.

MSP objects are configuration objects, which means that their formal object specification is again very simple:

```
public interface MultimediaStreamProtocol
   extends premo.std.part2.PropertyConstraint, java.rmi.Remote {}
```

The object specification also includes three properties:

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `NameK` | `String` | yes | Semantic name of the object. |
| `VersionNumberK` | `Integer` | yes | Implementation dependent value. |
| `ByteOrderK` | `String` | no | |

The purpose of the semantic name has already been explained. The version number allows for future revisions of MSP objects, in case new types of communication protocols come to the fore. Finally, the byte order property refers to the byte order in 16 bits integers. This property has a natural capability defined, too:

| Key | Type | Value |
|---|---|---|
| `ByteOrderCK` | `String[]` | `{"LittleEndian", "BigEndian"}` |

Although PREMO is not meant to give a detailed characterization of communication protocols, it does go one step further than the bare MSP objects by defining two subtypes of MSP:

1. `IntraNodeTransport` objects, which refers to communication among nodes taking place through shared memory (for example, two nodes processing in the same address space of a workstation, e.g., using DMA). In the case of Java, this refers to virtual devices sharing the same Java virtual machine.
2. `InterNodeTransport`, which refers to communication among nodes taking place over a network, or through some inter–process communication means. The various protocol names which can characterize these protocols include IPC, UDP, RTP, ATM, or NETBIOS.

Formally, the two objects are defined as follows:

```
public interface IntraNodeTransport
   extends MultimediaStreamProtocol, java.rmi.Remote {}
```

and

```
public interface InterNodeTransport
   extends MultimediaStreamProtocol, java.rmi.Remote {}
```

Furthermore, the semantic names of `InterNodeTransport` protocol objects are fixed through an extra capability for this object:

| Key | Type | Value |
|---|---|---|
| `NameCK` | `String[]` | `{"TCP","UDP","RTP", "ATM","BigEndian"}` |

It must be noted that, in the case of our prototypical implementation, where all inter–process communication takes place through Java RMI, the role of the various MSP objects is minor. Indeed, the Java RMI layer hides all the discrepancies which might occur between different machines, such as byte order. However, if a more general mechanism were used (for example, connecting Java JVM's to other processing entities directly through sockets), use of the MSP objects would become important.

### 6.2.3 Quality of Service Descriptor Objects

In order for a virtual resource to be useful, it must obtain the required physical resources required to do its job. Resources include both system resources that are typically not multimedia specific, such as those provided by the CPU, memory, and network subsystems, as well as specialized multimedia resources such as audio and video devices. Because the quality of service that can be provided by many resources varies considerably, the client must also specify the desired quality of service when requesting a resource. This is done by setting the properties of a special configuration object, called the `QoSDescriptor` object. Though quality of service can take on many meanings, many of them media and device specific, the MSS defines a core set of `QoSDescriptor` properties that can be used by a client to specify the desired quality of service.

Formally, the Quality of Service descriptor object is defined through:

```
public interface QoSDescriptor
   extends premo.std.part2.PropertyConstraint, java.rmi.Remote {}
```

The core set of properties are as follows:

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `NameK` | `String` | yes | Semantic name of the object. |
| `GuaranteedLevelK` | `String` | no | Indicates the required performance. |
| `ReliableK` | `boolean` | no | Is the delivery of data reliable or not? Is there a possibility of data loss? |
| `DelayBoundsK` | `Integer[2]` | no | Amount of time incurred between transmission of the data and its receipt. |
| `JitterBoundsK` | `Integer[2]` | no | Delay variance. |
| `BandwidthBoundsK` | `Integer[2]` | no | Minimum and maximum bandwidth. |

The guaranteed level property is an indication of the performance required on the connection. A `"Guaranteed"` connection will reserve resources to handle worst–case needs for the media transfer in order to make sure that the data always arrives and is on time. A `"BestEffort"` connection will provide the best possible performance while using optimistic amounts of resources. This may produce situations where the data occasionally arrives late. "NoGuarantee" uses the minimum amount of resources for the connection and do as well as possible.

Delay is the amount of time between the transmission of the data and the receipt of the data. Different applications will have different requirements. For instance, an audio conference would be unwilling to live with a 2 second delay, whereas a non-interactive video playback application might find it acceptable.

Jitter is the amount of delay variance. For example, an ISDN channel that presents a "slot" of data every 125 microseconds has a jitter of 0, since there is no variance in the arrival time of the data. If an application requests a jitter close to 0, then the connection will try to find an isochronous network connection between the two virtual devices.

Bandwidth is the amount of data per unit time that the connection will be required to support of, in the case of an input port, to expect. For example, a video conference might require 384 Kbits/second while an MPEG stream might require 1.5 Mbits/second. By defining the range of the bandwidth required, the connection will understand the minimum rate it must provide (in the case of an output port), or the minimum and maximum burst it must expect (in the case of an input port).

`QoSDescriptor` objects or, to be more precise, their subtypes defined by specific virtual devices, are supposed to define capabilities for all these properties, too. The capability for the `GuaranteedLevelK` property is defined by the PREMO document:

| Key | Type | Value |
|---|---|---|
| `GuaranteedLevelCK` | `String[]` | {`"Guaranteed","BestEffort", "NoGuarantee"`} |

whereas, for the delay, jitter, and bandwidth properties, the standard only requires that the range of allowed minimum and maximum values (e.g., for jitter bounds) should be defined through a capability.

Finally, two more capabilities are defined for `QoSDescriptor`:

| Key | Type | Value |
|---|---|---|
| `MutablePropertyListCK` | `String[]` | {`"GuaranteedLevelK", "ReliableK"`} |
| `DynamicPropertyListCK` | `String[]` | {`"DelayBoundsK", "JitterBoundsK, "BandwidthBoundsK"`} |

The keys for these capabilities are "inherited" from the `PropertyConstraint` object (see section 5.6.4 on page 115). What these capabilities mean is that the delay, jitter, and bandwidth bounds (which are all writable properties) may be changed by the client at any time whereas, once the media starts to flow (i.e., the `QoSDescriptor` object is "bound"), the client cannot change the performance and reliability properties. These two can be still changed by the "system" (e.g., by the virtual device), however. That is exactly what the notion of mutable property means. By setting an event callback to be triggered when these properties are changed (using the `setPropertyCallback` method, see page 71), the client can monitor the performance and reliability, though.

## 6.3 Stream Control

The issues and problems related to multimedia synchronization have already been addressed in section 5.5, which also introduced the object types `Synchronizable`, `TimeSynchronizable`, and `TimeSlave`. Theses objects serve as the basic building blocks for synchronization in MSS, too.

Synchronization concerns multimedia data, i.e., in terms of MSS, the flow of multimedia data through streams. In line with this "media flow" view, MSS defines two subtypes of the synchronizable objects. These extensions do not change the fundamental nature of the synchronization mechanism described in Chapter 5, they merely add more control over the flow of data.

Remember that in the specification of synchronizable objects "data presentation" is only an abstract notion (see page 94). The specification and the implementation of specific presentation techniques are left to subtypes. The specialization in MSS, vis–à–vis the "simple" synchronization objects, is to add more semantics to this data presentation step, while still retaining its abstract nature. The added semantics are as follows.

- Ability to "switch off" real data processing without suspending the flow of the media. This "mute" effect is well known from household devices; one switches off, for example, the sound on the TV, while the display still goes on. In terms MSS, this means that the media still flows through the streams, as if presentation really happened, but this has no sensible effect on the surroundings.
- Ability to temporarily "buffer" data. This means that, e.g., the incoming media data is put into a temporary buffer, instead of processing or forwarding it to the next device. Clients may use this facility if, for example, the quality of service degrades on the receiving end of a stream due to a high incoming load. Of course, facilities should also be provided to empty the buffer.

Much like configuration objects, stream control objects "belong" to virtual resources. What this means is that clients are not supposed to *create* instances of stream control objects. Instead, these objects are created by their "owner" virtual resources, and only their references are exported.

MSS defines two objects, called `StreamControl` and `SyncStreamControl`, to add these additional features to the synchronization model (see also Figure 6-4). These objects form the next elements of the puzzle in building up a virtual resource.

### 6.3.1 The `StreamControl` Object

Technically, the extensions, referred to above, are realized through an extension of the `TimeSynchronizable` object. The `StreamControl` object in MSS is a subtype of `TimeSynchronizable`, where the original state transition diagram (see Figure 5-12 on

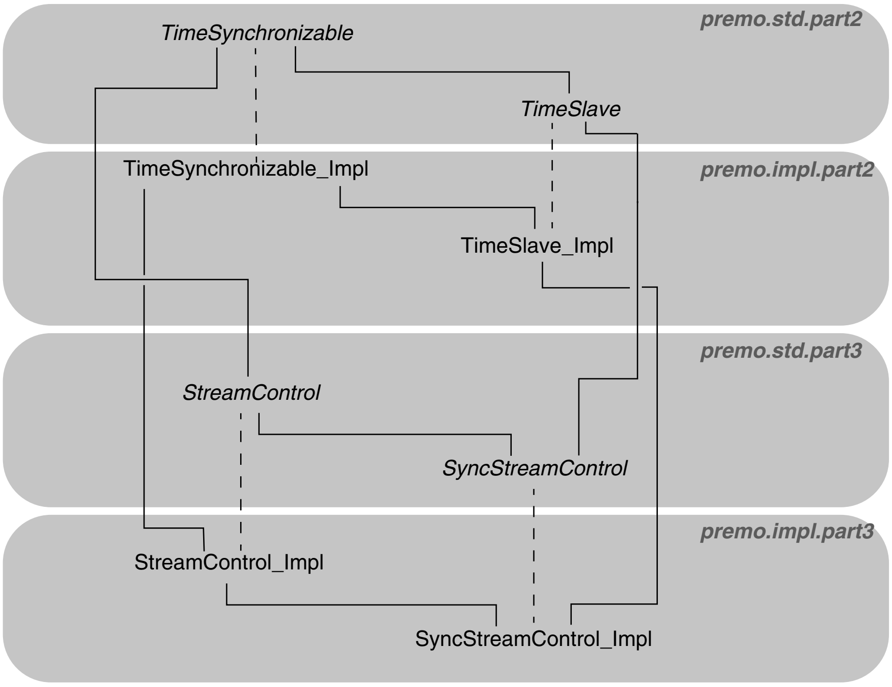


Figure 6-4 — Stream Control Objects

page 94) is modified. The new transition diagram is shown on Figure 6-5. Three new states are added to the state machine, namely `MUTED`, `PRIMING`, and `DRAINING`, together with new state transition operations.

The states `MUTED` and `PRIMING` are refinement states of the `STARTED` state of `Time-Synchronizable`. Refinement means that all three states (i.e., `STARTED`, `MUTED`, and `PRIMING`) behave identically in terms of synchronization. The additional semantics in the "new" states is only related to the notion of data presentation. Although the `Stream-Control` object does not specify what presentation means either (and leaves the details to the subtypes of `StreamControl`), the specification of `MUTED` and `PRIMING` gives a somewhat finer control on the behaviour of the `StreamControl` object. This refinement is as follows:

- No presentation occurs in `MUTED` state, i.e., the media data are disregarded. In other words, "progression" on the stream does occur (and all synchronization actions are performed) but the `processData` operation, which represents the abstract notion of processing media data, is *not* invoked.
- No presentation occurs in `PRIMING` state either, but the media data are buffered in an internal buffer instead of being simply disregarded. In other words, progression on the stream occurs (and all synchronization actions are performed) and the media data are stored internally instead of being presented, i.e., instead of calling the `processData` operation. If the internal buffer of the object is full (the "high mark"

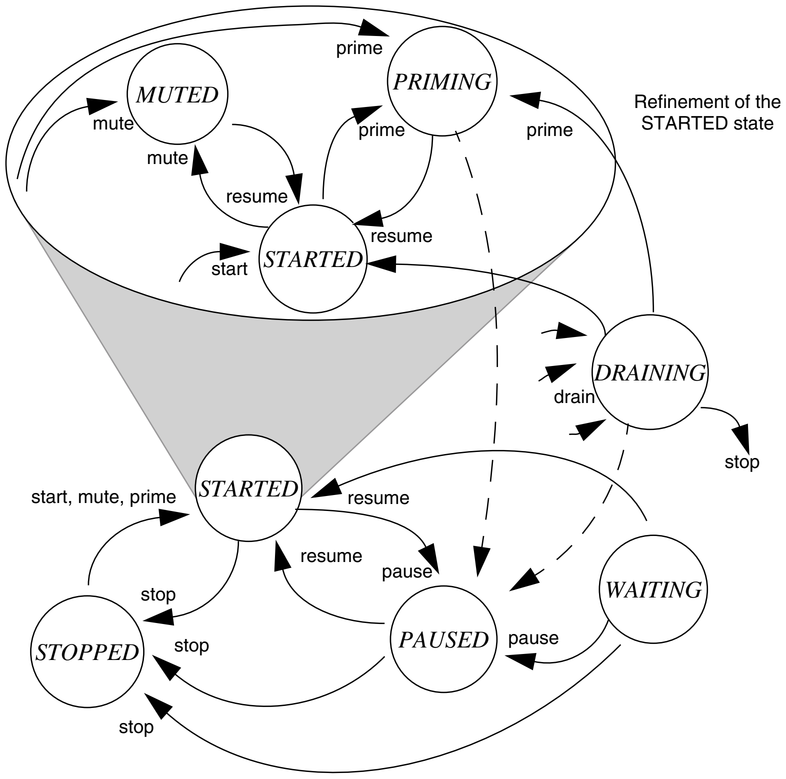


Figure 6-5 — State Transition Diagram for a *StreamControl* Object

is reached), then no stream data can be stored any more, and the object makes an internal state transition to PAUSED.

The third additional state, DRAINING, is the counterpart of PRIMING in buffer control. When set to this state, the object empties the buffer filled up by a previous PRIMING state. When the buffer is empty (i.e. the “low mark” is reached), the object makes and internal state transition to PAUSED. While emptying the buffer, presentation of data also occurs. The operation drain is defined to set the StreamControl object into DRAINING state.

As seen on the state transition diagram, a transition to STARTED state from both PRIMING and DRAINING states is possible through resume. Conceptually, the internal buffers are to be instantaneously emptied before the normal media flow is resumed.

Note that, as a subtype of Synchronizable, the StreamControl object inherits the ability to monitor state transitions (using the callback mechanism). In particular, clients can be notified if internal buffers become full or empty while priming, respectively draining (in both cases an internal state transition to the state PAUSED takes place, which may be monitored).

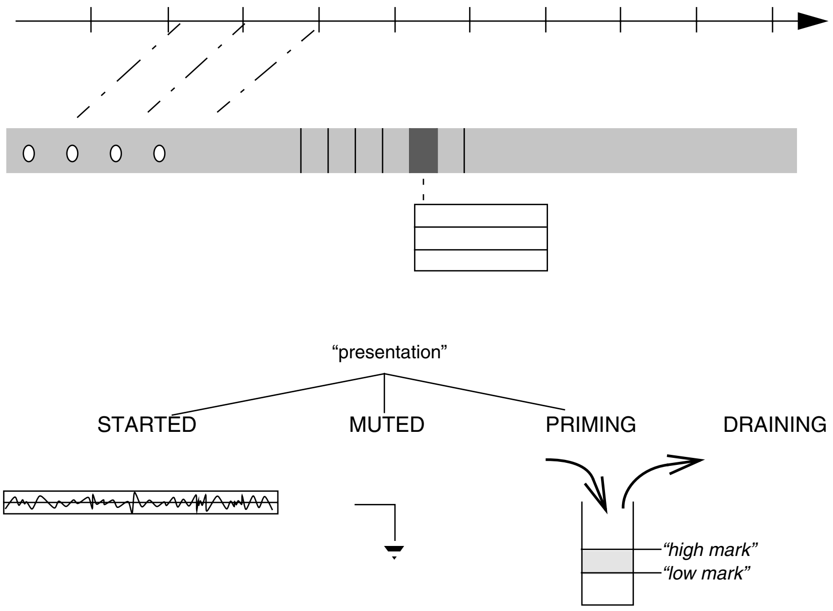


Figure 6-6 — States in a Stream Control Object

Subtypes of `StreamControl` may add additional semantics to buffer control. As a typical case, if the streams are aware of their position within a dataflow network, some of the operations, like `prime` or `drain`, may also generate private control flow among the stream control objects in this network. For example, `prime` on a `StreamControl` may also generate control information to the `StreamControl` object “up–stream”, i.e., the stream providing the data. Whether such additional protocol is defined or not depends on the subtypes of the `StreamControl` object and it is not defined by PREMO.

The Java specification of a `StreamControl` object’s interface is very simple:

```
public interface StreamControl
   extends premo.std.part2.TimeSynchronizable, java.rmi.Remote
{
   int getCurrentState();
   void mute()  throws WrongState;
   void prime() throws WrongState;
   void drain() throws WrongState;
}
```

i.e., the three new state transition operations are simply added. The operation `getCurrentState` is inherited from `Synchronizable`. In this case, it can return the integer codes for the three new states, too.

We close discussion on raw stream control with a note on the pragmatics of implementing and using this object type. Looking ahead, a key of stream control will be to underpin the management of media streams between devices, i.e., the progression space of a stream control object will be linked to specific media content. At this point the fact that the concept of time for a stream control object is relative time, i.e., time measured along the media content, not along the real time taken to deliver that content, becomes important. When the throughput of a media stream is reduced below a certain limit, such an object may choose to enter its PAUSED state, effectively suspending progression through the media content as well as time, and go back to STARTED state when data becomes available again. PREMO provides mechanisms through which the divergence that will result between this media–relative time, and any global measure of time, can be detected, e.g, through the monitoring of state transitions, through the monitoring of drifts in time measurements when time slaves and masters are involved, etc.

### 6.3.2 SyncStreamControl Objects

A SyncStreamControl object is used to permit the synchronization of *multiple* media streams. Although this sounds complicated, its specification is based on the various objects we already have defined. It is simply a common subtype of both a StreamControl object and a TimeSlave object (see section 5.5.3). The "multiple" aspect of synchronization is fulfilled by the semantics of the TimeSlave object. The Java specification is simply:

```
public interface SyncStreamControl
   extends premo.std.part2.TimeSlave, StreamControl, java.rmi.Remote
{}
```

The SyncStreamControl adds the time slaving facilities to StreamControl. Another way of putting it is that it extends the TimeSlave objects the same way as the StreamControl object extends the TimeSynchronizable object.

## 6.4 Virtual Resources

The "major" MSS objects are re–grouped under the virtual resource hierarchy (see Figure 6-7). The top level of this hierarchy is the VirtualResource object, which is an abstraction of a physical resource in a very general sense. The Multimedia Systems Services defines four basic subtypes of virtual resources:

1. *virtual devices*, which abstract media processors.
2. *virtual connections*, which abstract connections among virtual devices.
3. *groups*, which provide a convenient way to interact with a collection of virtual devices and connections.
4. *logical devices*, which make it possible to build a hierarchy of virtual devices.

These four subtypes represent very different semantics. There are, however, commonalities which make it worthwhile to re–group them into the same subtype hierarchy. The current section concentrates on these common features, whereas the rest of this chapter will describe each of the resource types individually.

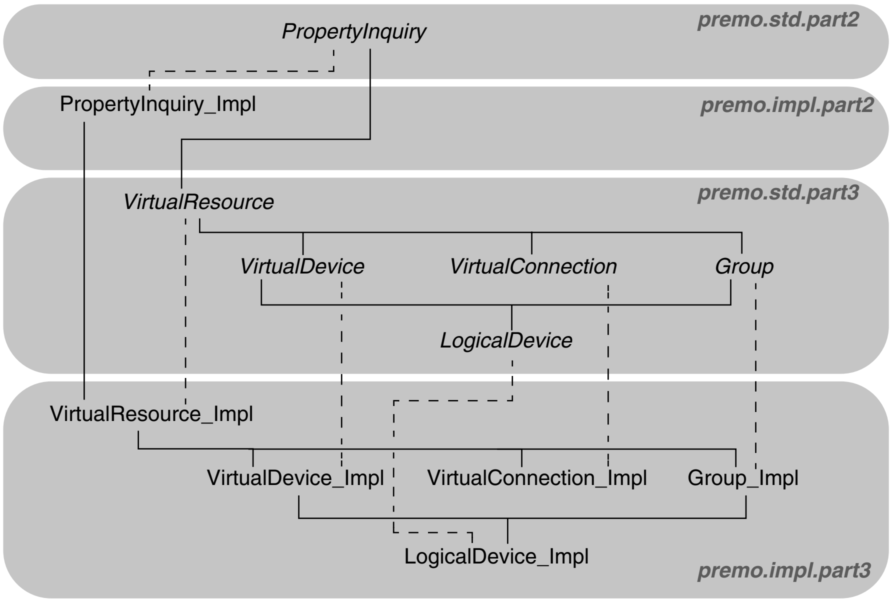


Figure 6-7 — Hierarchy of Virtual Resources

### 6.4.1 Property Control of Configurations

The `VirtualResource` object is a subtype of `PropertyInquiry`. However, and this is very important to note, a virtual resource object is *not* a subtype of the type `Property-Constraint`. In other words, a virtual resource object may have associated properties, capabilities, native property values, etc., which can be inquired and used, but the rich facilities to constrain properties are *not* at the client's disposal as far as virtual resources are concerned. Instead, configuring and adapting a virtual resource is done through separate `PropertyConstraint` objects. This is where configuration objects and their semantic names fit into the puzzle (see also page 130).

To achieve this, PREMO defines the tuple

```
public final class ConfInfo implements java.io.Serializable {
   public String  semName;
   public Class   objectType;
}
```

to characterize a configuration object: it contains the semantic name of the object, as well as its class (the Java `Class` object must refer to a subtype of `Format`, `Multimedi-aStreamProtocol`, or `QoSDescriptor`). Each virtual resource also has a property of the form:

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `ConfigurationNamesK` | `ConfInfo[]` | yes | Semantic names and types of all associated configuration objects. |

which lists all the configuration objects which are associated with the virtual resource. It is through *these* configuration objects that the virtual resource can be adapted to its environment, that it can be configured, etc.

Very often the list of semantic names is quite enough for the client, and it does not really need access to the object references themselves. We will see simple examples for this in later sections. However, if the client intends to more precisely configure the virtual resource, the real object references can also be accessed through the operation:

```
PropertyConstraint resolve(String semName) throws InvalidName;
```

(obviously, an exception is thrown if the name does not refer to a valid configuration object for that virtual resource instance).

Virtual resources, as subtypes of `PropertyInquiry` objects, are usually created through object factories (see section 5.7.1), which enables the client to choose among available virtual resources at run–time. For example if "`fact`" refers to a factory object instance, the following code fragment creates a device which is capable of handling images in GIF and/or PNG formats:

```
// Construct the two possible audio configurations: GIF and PNG
ConfInfo possible1 = new ConfInfo("GIF",Class.forName("GIFFormat"));
ConfInfo possible2 = new ConfInfo("PNG",Class.forName("PNGFormat"));
// Build a property pair for constraint. The key is
// ConfiguratonNamesK, and the value is the array of possible devices.
Object[] types      = new ConfInfo[] { possible1, possible2 };
PropertyPair request = new PropertyPair("ConfigurationNamesK",types);

// An array of constraints has to be created for the factory object;
// currently, there is only one constraint:
PropertyPair[] constraints = new PropertyPair[1] { request };
// The device is a fictious image displayer device:
Class devT = Class.forName("ImageDevice");
// Instruct the factory to create a device, which has a format for
// GIF and PNG (at least).The third parameter is null, indicating that
// no special initialization is required.
VirtualDevice theDev;
theDev = (VirtualDevice) fact.createObject(devT,constraints,null);
```

### 6.4.2 Resource and Configuration Management

The term "virtual resource" refers to the fact that these objects manage some sort of abstract resource. These resources can be very different from one another, examples include managing processing power, a shared file system, access to a display or an audio device. From the point of view of PREMO, these resources are purely conceptual entities, and the document does not contain any detailed specification for these.

From a client's point of view, however, the act of "acquiring" a resource must be made explicit. This is related to the strongly configuration–oriented behaviour of virtual resources in the MSS. Indeed, much like configuration objects, virtual resources must be treated differently when they are configured, i.e., when they are to be prepared to "acquire" a resource, then when the resource is already acquired, i.e., the virtual resource processes the media data. For that purpose, the `VirtualResource` interface defines two operations:

```
void acquireResource() throws ResourceNotAvailable;
void releaseResource();
```

which changes the state of the object.

Each configuration objects acts as a two–state finite state machine, controlled through the `bind` and the `unbind` operations. The effect of the `acquireResource` operation for a virtual resource is twofold:

1. it tries to acquire the resource associated with the object, e.g., get hold of the audio device.
2. it invokes the `bind` operation for *all* configuration objects associated with the virtual resource.

Obviously, the role of `releaseResourse` is to invert these actions, including the unbind action on all configuration objects. The state of the virtual resource can also be inquired at any time. The operation

```
ResourceState getResourceState();
```

returns a PREMO enumeration instance of the type:

```
public final class ResourceState
   extends premo.impl.utils.PREMOEnumeration
{
    public static ResourceState ACQUIRED;
    public static ResourceState RELEASED;
}
```

As we have seen in section 5.6.4 (on page 117), the `bind` operation, defined on a `PropertyConstraint` objects (i.e., for all configuration objects), also performs a dependency check on the properties defined on a single object, detecting mutually incompatible property value settings. The same incompatibility problem might occur for a virtual resource, too, although on a higher level: are the combination of configuration objects and their respective properties, as set by the client, viable? To handle this situation, yet another operation is defined for a virtual resource interface, namely:

```
public class ProposedValues implements java.io.Serializable {
   public String          semanticName;
   public PropertyPair[]  replacement;
}
public class ValidationResult implements java.io.Serializable {
   public boolean         result;
   public ProposedValues  proposedValues;
}
ValidationResult validate();
```

Although the definition is a bit convoluted, the intention is straightforward. If the validation reveals no problems, the `result` field of the output structure is `true`. Otherwise a "replacement" for the property values is proposed: for each configuration object (identified with its semantic name) a sequence of acceptable property pairs is returned. Using these values, the client can be sure that the properties of the configuration objects will be set to optimal values. Much like the `select` and `constrain` operations defined for each configuration object, the interface of the `validate` operation may hide a significant complexity in finding optimal these values.

It is worth, at this point, to summarize all the various configuration steps the client may want to use for a virtual resource. This illustrates the rich configuration facilities offered by PREMO. Here are the possibilities a client may use:

1. A factory is accessed through a factory finder. The access of a factory may be controlled through constraints on the type of objects the factory may create.
2. A virtual resource is generated through a factory. This generation may be constrained through properties, e.g., through the choice of specific configuration objects.
3. The client may inquire the list of configuration objects available for the virtual resource instance and, if applicable, may select among those (details of this step will be described in later sections).
4. For each configuration object the client may
   – set/retrieve individual property values.
   – properties may be examined through the `matchProperties` operation.
   – properties may be constrained to some specific values through the `constrain` operation.
   – the configuration object may be instructed to set the best–fit values, depending on its local knowledge about its environment (through the `select` operation).
5. The virtual resource object can validate the property values through the `validate` operation, and possibly suggest a replacement for some key–value pairs.
6. If the client is satisfied with all the values, the resources are "acquired", which will also put all configuration objects into bound state.

Clearly, the various possibilities offered by PREMO can be used for very complex configuration procedures.

### 6.4.3 Stream Control

Typically, virtual resources are involved in the generation, consumption, or transport of media data. As we have already seen, the flow of media data through a device or across a connection can be thought of as a stream. The synchronization and the buffer control of this stream can be achieved using the stream control objects introduced in sections 6.3.1 and 6.3.2. To achieve this control, a virtual resource object has a global stream control object. This stream control object is created and contained by the `VirtualRe-`

`source` object instance (i.e., this object is *not* created by an external client). The reference to this overall stream control of the virtual resource can be retrieved through the operation:

```
StreamControl getStreamControl();
```

which is defined for the `VirtualResource` interface.

The semantics of stream control is very much dependent on the exact nature of the virtual resource, on its behaviour and its role in a network. Subsequent sections will provide more details on this. For some virtual resources the notion of global stream control is not even meaningful, e.g. `getStreamControl` returns `null` in this case.

### 6.4.4 Monitoring Resource Behaviour and Quality of Service Violations

In an ideal world, once the resources are set and the media data is flowing, the client may just “sit back and watch”. However, in a real world, resource availability may fluctuate, speed or space problems may occur. The client must be notified about these to implement some emergency measures.

One of the common problems which might happen is to loose the resources (e.g., the network is down). To monitor this situation, the client may attach a callback to each `VirtualResource` object through the operations:

```
void     setResourceEventHandler(Callback e);
Callback getResourceEventHandler();
```

If such a callback object has been assigned to the virtual resource, and the resource is lost, an event will be raised through this callback, which can be caught by the client. The name of this event is “`ResourceLost`”. If, subsequently, the resource is acquired again, another event is raised named “`ResourceAcquired`”.

Unfortunately, losing and acquiring resources represent only the two extremes of the spectrum. A finer control of resources, and the reaction to the changes in the availability of these resources, is a more complex issue, commonly referred to as Quality of Service (QoS) control. Procedures for maintaining specific quality of service characteristics of media flow are still the subject of active research. Perhaps not surprisingly by now, PREMO does not define any detailed policy for QoS management. This is left to clients (it would not make sense to impose one approach over the other). The general callback is used by the virtual resource to notify the client about any problem which might occur (in fact, subtypes of virtual resources define additional callbacks for specific purposes, but this “global” callback is always available).

As we have already seen, a specific configuration object type, the `QoSDescriptor`, is used to set QoS requirements for a resource. An instance of such a descriptor object is available for a virtual resource. PREMO specifies that a virtual resource object must raise an event through its global callback if a violation of these requirements occurs. This event has the name `"QoSViolation"`.

- The event data should contain the key–value pairs (i.e., the properties) whose requirement have been violated.
- If the QoS management is attached to a port of a virtual device, an identification of the port is also attached to the event data.

Specific subtypes of `VirtualResource` may add additional data to this event. Note also that it is not specified by PREMO that this global callback should be used for QoS management purposes only. Subtypes may define additional use of the very same callback object. However, by using the constrained registration mechanism for event handlers (see section 5.4.1.2), special QoS agents can be created which react only to QoS violation events.

## 6.5 Virtual Devices

Virtual devices are the “core” elements of MSS. They are the nodes in the multimedia dataflow network, they embody the processing, capture, or display of multimedia data. Virtual devices are roughly analogous to what other multimedia systems often call “media” objects.

Virtual devices are extensions of virtual resources and, as such, they inherit the facilities described in the previous chapter. They have a global stream control, a set of configuration objects, callback facilities. Specialized devices can easily add media–specific semantics to these general concepts, and this chapter will also specify how these global configuration objects can be set by the client. The major extension of the virtual device interface is the existence of ports, which enable virtual devices to communicate with other devices in a network.

The remainder of this chapter on virtual devices is divided into two parts. The configuration of devices, i.e., their run–time adaptation to a complete multimedia network, is described first. In fact, the *interface* specification of virtual devices in MSS is exclusively concerned with configurability.

The more “semantic” aspect of virtual devices, i.e., their internal organization, functioning, etc., is not detailed in the PREMO document, simply because the various subtypes represent a large variety of possible devices (for example, in Part 4, PREMO defines a range of different virtual devices adapted to modelling and rendering). However, examples of how specific devices may be realized should help in understanding the possibilities. Such examples are presented in the second half of this chapter.

### 6.5.1 Configuring Devices

Device configuration follows the approach already described in the previous chapters on configuration and virtual resource objects. The interface of virtual device make it possible to put this into practice, in terms of explicit operations.

Devices may be configured on two levels: through the “global” (i.e., port independent) aspects of the device (these are, essentially, inherited from `VirtualResource`) and through the port specific details. These are described separately.

#### 6.5.1.1 Global Configuration

The virtual device object defines the following operations for global configuration:

```
void          setResourceEventHandler(Callback e);
Callback      getResourceEventHandler();
void          setConfigurations(ConfInfo[]);
ConfInfo[]    getConfigurations();
```

The first two operations are inherited from `VirtualResource`, and are listed here for completeness only. The global (i.e., port independent) configuration objects are set and retrieved through the `setConfiguration` and `getConfiguration` operations, respectively. These configuration objects (such as `QoSDescriptor` objects) are used by the virtual device for, e.g., monitoring quality of service violations (see section 6.4.4). The exact semantics of their use depend on the specific subtypes.

A property is also defined for each virtual device of the form:

| **Key** | **Type** | **Read–only?** | **Description** |
|---|---|---|---|
| `GlobalFormatTypesK` | `ConfInfo[]` | yes | Types of configuration objects which can be assigned on a global level. |

which informs the client which configuration objects can be used globally. A capability is also defined:

| **Key** | **Type** | **Value** |
|---|---|---|
| `GlobalFormatTypesCK` | `Class[]` | `{QoSDescriptor, Format}` |

which tells the client what configuration types are available for global settings. Subtypes may impose further restrictions on this capability.

### 6.5.1.2 Port Configurations

As said before, *ports* of a virtual device are standard "openings", through which media data can flow in or out a device. As shown on Figure 6-2 (page 127), a virtual device, while retaining the global control and configuration objects, has additional object instances assigned to specific ports. To simplify the management of these objects, PREMO defines a separate structure called `PortConfig`, which is used as a tool to configure individual ports.

#### *6.5.1.2.1 Port Configuration Structures*

The fundamental data structure, which is used in port configuration, is defined as follows:

```
public class PortConfig extends SimplePREMOObject
{
   public ConfInfo      qos;
   public Callback      eventHandler;
   public ConfInfo      protocol;
   public static class formatData {
      public long       time;
      public ConfInfo   name;
   }
   public formatData[]  formats;
   public StreamControl streamControl;
}
```

[Note: The `ConfInfo` structure has already been defined on page 141. It is a tuple of a semantic name and a `Class` instance for a configuration object]. This structure groups all the objects which are relevant for the configuration of a port. None of these objects are really different from what has already been described for general virtual resources, but it is still worth going through each of them in more detail:

- The `qos` object refers (through the `ConfInfo` tuple), to a `QoSDescriptor` object (see section 6.2.3). Its role is to act as a depository for Quality of Service requirements *on a specific port.*
- The `eventHandler` object is used by the virtual device if problems occur which are specific to that port. Its primary use is to provide a handle for quality of service monitoring on the port, much the same way as monitoring global QoS violation, described in section 6.4.4.
- The `protocol` object refers to an instance of a `MultimediaStreamProtocol` object (see section 6.2.2). Whereas the quality of service and format objects may have a general meaning for the device at large (i.e., as global configuration objects), MSP objects are typically useful on a port only, being intimately related to external connections.
- `formats` is an array of `Format` objects (see section 6.2.1), referred through the respective `ConfInfo` structures, and ordered in time. This means that the client can not only control and assign specific formats to a port, but can also set a validity interval for a format, i.e., have the formats change over time. The value of the time is related to the flow of time on the port's stream control object.
- The `streamControl` object is the entry point for synchronization on the port level. As a subtype of `TimeSynchronizable`, it also has its own timeline, with settable time units, used to order the `Format` objects in the `formats` array in time.

Just as in the case of global configuration and stream control objects, none of the objects in the port configurations are created by the client. Instead, the virtual device instance creates them, and the client can access their semantic names or references.

#### *6.5.1.2.2 Configuring Ports*

Each port of a device is identified by an integer, referred to as a port identifier. The operation

```
int[] getPorts();
```

may be used to retrieve all available port identifiers. Using a port identifier, the client can retrieve the current port configuration through the operation:

```
public static class PortDescr
{
   public PortConfig   config;
   public PortType     type;
}
PortDescr getPortConfig(int portId) throws InvalidPort;
```

(The exception `InvalidPort` is thrown, obviously, if the `portId` value does not refer to a valid port of the device.) The `PortConfig` structure has been described in the previous section; `PortType` is a simple PREMO enumeration of the form:

```
public final class PortType
   extends premo.impl.utils.PREMOEnumeration {
    public static PortType INPUT;
    public static PortType OUTPUT;
}
```

The port configuration can also be set by the client, using the operation:

```
void setPortConfig(PortConfig config)
   throws InvalidPort, InvalidName, InvalidPosition;
```

Obviously, exceptions are raised if the content of the config object is invalid (e.g., one of the configuration objects cannot be assigned to the port, the time value used in the `Format` array is invalid, etc.).[1)]

We have already described, on page 144, the general configuration facilities the client has at its disposal when dealing with MSS objects. These facilities involve, to a large extend, the property management mechanism on configuration objects. Port configurations add another level of configurability to MSS, insofar as a fine control over the configuration objects on a port by port basis becomes possible.

Additional properties are also defined for virtual devices to help the client in its configuration task. A separate data type is necessary for these properties, namely:

```
public class PConfInfo implements java.io.Serializable {
   int        portId;
   ConfInfo   config;
}
```

which is simply a tuple of a port identification and configuration object description. Using this structure, the corresponding properties are:

[1)] There is, actually, an awkwardness in the PREMO specification at this point: although the `PortConfig` class specification includes a reference to the port stream control object, too, the client is not supposed to change that reference when setting the port configuration.

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `InputPortK` | `int` | yes | Number of input ports. |
| `OutputPortK` | `int` | yes | Number of output ports. |
| `InputFormatK` | `PConfInfo[]` | yes | Types of configuration objects which can be used in conjunction with specific input ports. |
| `OutputFormatK` | `PConfInfo[]` | yes | Types of configuration objects which can be used in conjunction with specific output ports. |

Capabilities are also defined:

| Key | Type | Value |
|---|---|---|
| `InputPortCK` | `int` | Maximum number of input ports. |
| `OutputPortCK` | `int` | Maximum number of output ports. |
| `InputFormatsTypesCK` | `Class[]` | Allowed configuration objects on input ports. |
| `OutputFormatsTypesCK` | `Class[]` | Allowed configuration objects on input ports |

Note that the real values for these capabilities are defined in the specific subtypes of virtual device.

The virtual device inherits the `validate` operation from virtual resource (see page 143). This operation checks whether the current combination of configuration objects are "valid", i.e, the properties on these configuration objects do not lead to conflicting requirements. Because the configuration of a virtual device is also done on a port by port basis, an additional operation is necessary to restrict this kind of check for a port. This operation, called `portValidate`, checks whether a specific `Format` object is "compatible" with the quality of service and protocol requirements, as set by the configuration objects in the port configuration structure. If this is not the case, the operation returns a proposed "replacement" for the property values for these configuration objects, which might make the requirement on the port viable.

The formal specification of this operation involves the same, somewhat complicated inner classes of `VirtualResource`:

```
public class ProposedValues implements java.io.Serializable {
   public String          semanticName;
   public PropertyPair[]  replacement;
}
public class ValidationResult implements java.io.Serializable {
   public boolean         result;
   public ProposedValues[] proposedValues;
}
```

Using these classes, the specification of `portValidate` is:

```
ValidationResult portValidate(int portId, String formatName)
   throws InvalidName, InvalidPort;
```

## 6.5.2 Examples of Virtual Devices

The PREMO document does not stipulate any specific architectural approach for the implementation of virtual devices. As long as an object implements the interface detailed in the previous chapters, i.e., the device is properly configurable, this object can be considered PREMO compliant. The term “Processing Element” appearing, for example, on Figure 6-2, is purely a conceptual entity, just like a port, which does not have any interface specification.

One of the major design decisions, when planning for a new device, is the division of control and work among the various stream control objects: the global stream control, and the stream control objects assigned to the ports. There are cases (we will see some examples below) when these object references refer to the very same object, i.e., when there is, in fact, only one stream control object in the device. However, this is not always the case.

Although MSS does not specify in detail what the relation among these control objects is, it does say that the client should be able to focus all inquiry and control methods concerning data stream at the global stream control object. The role of the stream control objects on the ports is therefore slightly less important, and becomes significant only when very fine grained synchronization is necessary. This “priority” of the global stream control object is reinforced by an additional requirement formulated in MSS: this requires that all port related stream control objects must be subtypes of the global stream control. In the Java case this means that, for example, if the virtual device implementation defines an `MStreamControl` interface, which extends the `StreamControl` interface, and the the global stream control implements `MStreamControl`, then all port related stream control objects must also implement the `MStreamControl` interface.

This chapter gives some examples for virtual devices. None of these are described in the PREMO document in detail, but they are all possible incarnations of the general virtual device concept. Our prototypical implementation of PREMO implements some of these, too.

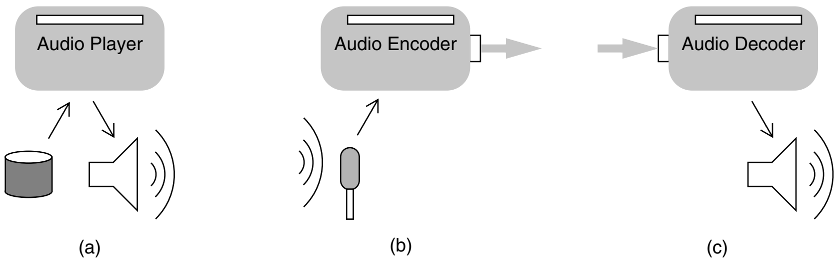


Figure 6-8 — Simple Virtual Devices

### 6.5.2.1 Simple Media Devices

Although a large portion of the previous chapters concentrated on how device ports are set up and configured, it should be emphasized that a device without a port is a very important concept by itself, too. As an example, consider a simple audio device which reads an audio file directly, and plays it (see Figure 6-8/(a)). Because no media data transfer occurs between this device and other devices, such a device would indeed have no ports. Nevertheless, the device can be configured for its format (through the general configuration objects), possible quality of service violations can be monitored (through the interplay of `QoSDescriptor` objects and the callback facilities, see section 6.4.4 above), resource allocation can be controlled (resource allocation would mean, for example, finding and opening the audio file, for example) and, last but not least, synchronization facilities are available through the global `StreamControl` object of the device (the global stream control object is depicted by the white stripe on the figure). A way of looking at such a device is to say that it is a configurable wrapper around a `StreamControl` object (i.e., a `Synchronizable` object), whose progression (see section 5.5.1 on page 92) results in the display of the media data. Control over this progression (i.e, start, stop, pause, resume, mute, etc.) gives control over the media display. The “processing element” in this case is virtually identical to the active entity controlled by the `Synchronizable` object, and which is responsible for the details of progression and media presentation.

Figures 6-8/(b) and 6-8/(c) show two complimentary devices. The first device (the audio encoder) receives audio data directly from a microphone, and turns the data into an audio format suitable for the multimedia network. The audio decoder, on the other hand, receives audio data on its input port, and plays it. The only functional difference between this device and the audio player is that the audio data arrives at the device through the network and not directly from a file. Both the audio encoder and decoder have a very similar structure to the audio player. They have global control for synchronization, control over progression, etc. Because there is only one input port, a sensible choice of the device implementation is to define only one stream control object for the device, i.e., the global stream control and the one which is part of the port configuration would refer to the same object.

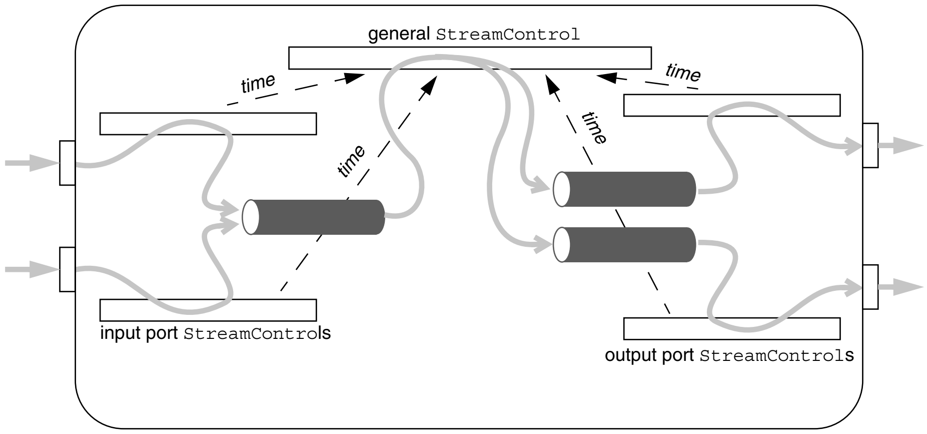


Figure 6-9 — Inner Structure of the `Transformer` Object

Obviously, the audio encoder and decoder objects could be “piped” together by using a virtual connection to connect their ports (see section 6.6 below for the details of virtual connections). The port configuration process, described in earlier chapters, should make it sure that the audio format produced by the audio encoder is understood by the decoder. Whether this connection is done within the same machine, joining the microphone and the speaker on the same workstation, or whether there is a network between them, is immaterial. The functional behaviour of both devices remain unchanged (of course, the level of service quality might become different, and the client should monitor this!).

Devices become more complex when they have both an input and an output port, e.g., when they act as filters or format converters. The “processing entity” is then responsible to read the data from the input port, convert the data and send it to the output port. Instead of going into the details of how such a device could be implemented, we will present a slightly more general device type, which is a good example for a large family of virtual devices.

### 6.5.2.2 Transformer Devices

`Transformer` is the name of a virtual device type, developed in the course of our prototype implementation. This device has *not* been defined as part of the PREMO document (it is not a standard object), but it has proven to be a useful abstraction to help implement a whole range of other devices, such as the devices defined in Part 4 of PREMO (see Chapter 7). Although, throughout the book, we restricted ourselves to presenting only standardized PREMO objects in detail, we make an exception here. Indeed, through the `Transformer` object we hope the reader will gain a better appreciation of how virtual devices may be implemented. The goal of the `Transformer` object is to act as a supertype for media filters and media processors for multimedia data. Format converters are obvious examples. Other examples include wrappers around database facilities, image filters, geometric processors, etc.

Figure 6-9 shows the internal structure of a `Transformer` object (to simplify the figure, the configuration objects are not shown). The complexity of the implementation comes from the exact distribution of work among the possible stream control objects represented by the white stripes on the figure. Obviously, there are $n + m + 1$ stream control objects in a `Transformer` object, where $n$ is the number of input, and $m$ is the number of output ports, respectively. To be more precise, the control objects assigned to the ports are all `SyncStreamControl` objects in this case, i.e., their timelines may be slaved. In a `Transformer`, they are all slaved to the general stream control object, which may or may not be of `SyncStreamControl` type. This means that all stream control objects in a `Transformer` refer to the same clock for synchronization purposes. The availability of the `TimeSlave` interface (see section 5.5.3 on page 107) for the port stream control objects allows the client to monitor any drift in time between this object and the general stream control, thereby detecting possible starvation due to upstream congestion or failure.

Stream control objects (as `TimeSynchronizable` objects) are all active entities. In the Java case, they all run in their own thread of control. It is the interplay of the various threads which build up the functionality of the object. The central role is played by the global stream control object. Another way of putting it is that the abstract processing unit in Figure 6-2 essentially can be identified with the thread of control of the global stream control object.

A stream control object assigned to an input port transfers the input data into a (multiplexed) data queue. Each piece of datum is stored in this queue together with a tag denoting its “origin”, i.e., the port it arrived from. The input port stream control does not do any processing, its role is merely to transfer data (using the terminology for the general `Synchronizable` objects, its progression space is the sequence of input data units, and its “presentation” step is simply copying data into the internal queue). The speed of this transfer is controlled by this object (using its `speed` attribute), it can react to a `mute` operation request by simply ignoring the data, and, more importantly, it can monitor the quality of service of the data transfer itself (e.g., by monitoring its own clock, it can generate an error if the data-rate on a particular port doesn’t meet the bandwidth requirement). If callbacks are set for quality of service violations, the object will raise the necessary events.

The global stream control object is the real “worker”. It sequentially reads the data from the multiplexed input queue (which plays the role, in a sense, of the progression space of this object), it transforms the data, and puts the results into separate queues which connect the general stream control object to output ports. Of course, on the level of the `Transformer` object, some details of this step are left abstract. The exact nature of processing and the choice of the target output ports are left to subtype implementations. While doing the transformations, the object can react to all synchronization requests. This is inherited from the behaviour of `Synchronizable` objects (as said before, the general stream control is the focal point of the device for multimedia synchronization). Finally, the role of the output stream control objects is the counterpart of the input stream control objects: they read the data from their queues to forward them to the output streams.

Depending on the exact nature of the `Transformer` subtype, the state transitions of the global stream control, as induced by an external client, may internally change the states of the port stream control objects, too. For example, if the global stream control object is stopped, all stream control objects are to be stopped, too. If it is muted, this should "mute" the output stream control objects to avoid transferring data to the output ports.

Of course, lots of details are not addressed here, such as how to optimize the architecture to avoid *busy waiting* in the various threads, how to avoid memory overflow, etc. However, this short overview of the `Transformer` objects will hopefully help the reader to have a better understanding of the issues involved in virtual devices.[1)]

## 6.6 Virtual Connections

### 6.6.1 Overview

So far, the media stream has always been presented as a purely abstract entity. In practice, streams are realized through communication facilities which connect the active virtual devices. Although, conceptually, it is the role of virtual devices to "move" media data, an additional mechanism should be provided which transfers the data from the output port of a device to the input port of another. There may be a large variety of manifestations of this mechanism, depending on the implementation environment PREMO runs in. It may use network facilities, remote procedure calls, remove objects, shared memory, etc. It is obviously not the role of PREMO to give a detailed specification for all these various communication facilities. There are numerous standards and packages that do this already.

However, the client needs a focal point to set up and to dismantle such a connection. Although the details of the communication mechanisms may widely vary between PREMO implementations, a PREMO application still needs to have a unified view and control over communication. This is the role of the `VirtualConnection` object. Through an instance of a virtual connection object, a client may set up a connection between two ports, and may also disconnect the ports when the streams are no longer necessary. Because a virtual connection is also a virtual resource, the QoS characteristics of the connection can also be monitored, which may be of great importance to the client.

The virtual connection also plays an important role in the proper modularization of virtual devices. Two virtual devices, connected by a stream, might run in a variety of settings: they may share the same virtual or physical processor, or they may run on different nodes of a local or wide area network. They may communicate through shared memory, ATM, TCP/IP, etc. However, the implementation of a specific virtual device should *not* depend on its particular position in such a web of communication. It should be able to write or read data *regardless* of the mechanism which physically delivers the data itself. It is the role of the virtual connection to hide these specific details, and to

[1)] It is interesting to note that the internal structure of the `Transformer` bears a lot of similarities with the basic processing blocks described in the ISO Computer Graphics Reference Model[47]. Similar functionalities lead to similar architectures…

provide a unified interface to the virtual device. Details are of course implementation dependent and are invisible to the client. As an example, we will show later how this feature was achieved in our prototypical implementation.

Beyond the "physical" communication, connecting two device ports also involves an "agreement" between the ports regarding the media data format they transfer. Finding this "agreement" is also the task of a virtual connection. In other words, a virtual connection embodies some of the configuration tasks which characterizes PREMO. Various PREMO implementations may differ on the quality of configuration characteristics they can provide and, of course, a PREMO application may also define its own subtypes of the `VirtualConnection` objects to adapt them to their needs.

### 6.6.2 Detailed Specification of Virtual Connections

Formally, the interface of a virtual connection object is defined as follows:

```
public interface VirtualConnection
   extends VirtualResource, java.rmi.Remote {
   void connect(  VirtualDevice master,   int portMaster,
                  VirtualDevice slave,    int portSlave)
      throws   ConfigurationMismatch, PortMismatch,
               ResourceNotAvailable, InvalidPort;

   void disconnect();

   public class EndpointInfo {
      public VirtualDevice device;
      public int           port;
      public boolean       isMaster;
   }
   EndpointInfo[] getEndpointInfoList();
}
```

Obviously, it is the `connect` operation which requires most of the explanation. The client calls this operation to create a connection between two ports, identified as master and slave ports, respectively. The term "master" does not mean that it is necessarily an output port. It means that, in the course of the configuration steps, it is the configuration belonging to the master which prevail, if a choice must be made. For example, if both ports can manage "`LittleEndian`" and "`BigEndian`" byte orders (as specified through the properties with key "`ByteOrderK`" in the `MultimediaStreamProtocol` object instances assigned to the port), but the `select` operation of on the master's side yields "`LittleEndian`" (meaning that this is preferred format of the master), then this value will be chosen.

Setting up the connection involves the following steps.

1. The exact format of the media flow may have to be negotiated. To achieve this, the virtual connection object inquires the available `Format` objects on both ports to find appropriate matching pairs. We have already seen a more detailed example on page 131 on how the various formats can be matched. These actions may be carried out by the virtual connection object.

It is *not* required, however, that only virtual connections perform such configuration steps. In some cases, the client can perform a much finer configuration setting based on its own application semantics. Consequently, subtypes of virtual connections can be defined by an implementation which relies on configuration being performed by the application.

2. A communication mechanism has to be set up between the ports in the "direction" dictated by the output and input ports (a `PortMismatch` exception is thrown if, for example, both ports are input ports). In contrast to the previous step, clients usually have no real control over how this is done; the details are deeply rooted in the implementation environment of PREMO. The virtual connection may consult, however, the MSP objects assigned to both ports, to decide upon the best communication channel.
3. Quality of service requirements are set for the connection. The virtual connection object is also a virtual resource, which means that the client may set quality of service requirements for the connection through its `QoSDescriptor` objects. Furthermore, similar configuration objects are available on both the master and the slave ports. Based on this information, the virtual connection object may set up its own quality of service management, which will influence, for example, when a QoS violation event will be raised. The three `QoSDescriptor` objects may also impose restrictions which cannot be fulfilled by the connection instance. A `ConfigurationMismatch` exception is thrown in this case.

Subtypes of `VirtualConnection` may add additional connection control, of course.

Formally, the `connect` operation only sets up the connection, which is not necessarily "alive" yet on return from the `connect` operation. This distinction is important if, for example, the connection involves some active entities, e.g., separate processes or threads, which have to be activated separately. This activation is done through the `acquireResource` operation, inherited from `VirtualResource`. When releasing the resources, these active entities may also be suspended.

A single `VirtualResource` instance can manage only one connection at a time. The `ResourceNotAvailable` exception is raised by the `connect` operation if the client attempts to connect a pair of ports without disconnecting the previous one. The `Group` object (see section 6.7) can be used to group several `VirtualConnection` object. This may be necessary, for example, in order to "synchronize" the `acquireResource` operation on all of them.

### 6.6.3 Examples of Virtual Connections

It is worth looking at some examples of connection setups, to make the role of a `VirtualConnection` object clearer. We will concentrate on the details of step 2 on page 157, i.e., on the physical connection being set up between two ports.

In the first example, shown on Figure 6-10, a separate buffer is necessary to keep up, e.g., with the quality of service requirements of the client. The virtual connection may then set up a separate buffer manager (invisible to the client) and, although data concep-

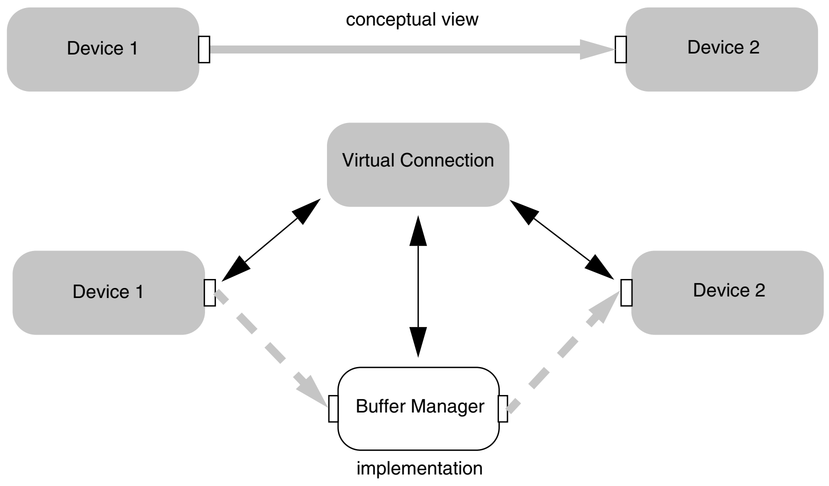


Figure 6-10 — Buffered Connection

tually flows from Device 1 to Device 2, the real data flow is from Device 1 to the buffer manager, and from the buffer manager to Device 2. The buffered manager itself is an active object which has to be activated by the virtual connection explicitly when its resources are acquired.

The second example, on Figure 6-11, shows what may happen if the two virtual devices run on two distinct machines. Depending on the characteristics of the communication facilities, the virtual connection may have to instantiate two "gateway" processes, whose role is to forward the media data from one system to the other. Of course, these gateway processes are invisible to the client.

The task of implementing a virtual connection is much easier if the implementation environment provides advanced networking facilities already. For example, our prototypical implementation makes use of the fact that all our objects run in a homogeneous (i.e., purely Java) environment, and that the standard `java.net` package takes care of most of the local specificities of communication (for example, byte order). Figure 6-12 shows the solution we have adopted to connect two ports in our prototypical implementation. When asked to connect two ports, the implementation of the `VirtualConnection` object shown in Figure 6-12 does the following:

1. The `VirtualConnection` checks whether the two virtual devices are running within the same Java Virtual Machine. If yes, a pair of `PipedOutputStream` and `PipedInputStream` objects are used to connect the two ports (both these objects are part of the `java.io` package).

2. If the two virtual devices are running on different Java Virtual Machines, the connection establishes a dedicated socket pair between the two JVM's (facilities are provided by the java.net package). The `java.net.Socket` object also provides stream access to these sockets in the appropriate direction, which are used to connect these sockets to the ports.

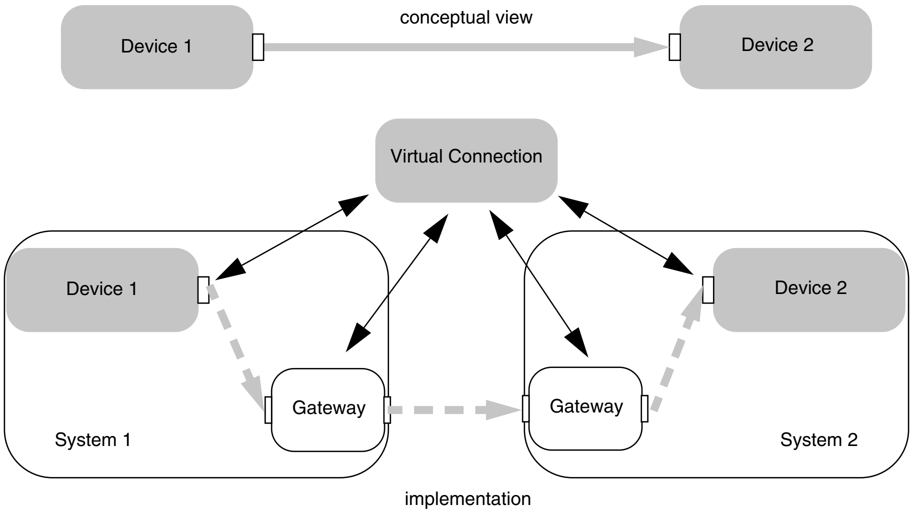


Figure 6-11 — Networked Connection

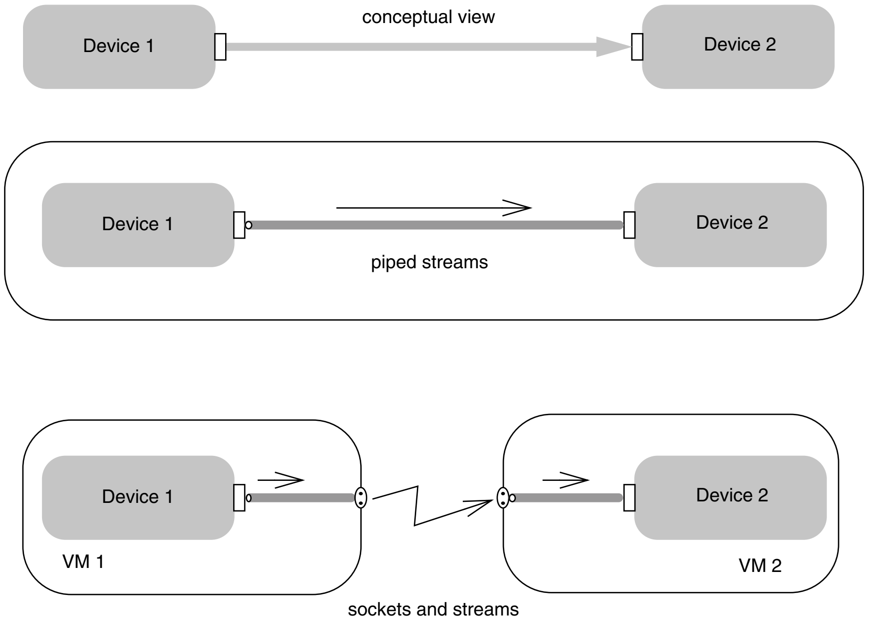


Figure 6-12 — Connections in Java

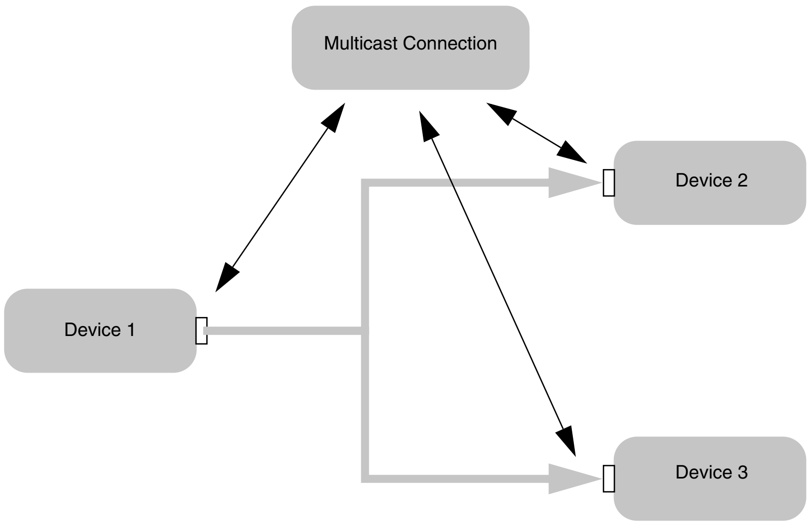


Figure 6-13 — Multicast Connection

In both cases, the streams may be combined with the buffered output and input streams of `java.io`, if necessary, i.e., a separate buffer manager, as shown on Figure 6-10, is not necessary[1]. As a result of this construction, virtual device objects have access to standard Java `Stream` objects only (not to be confused with PREMO Streams!) when moving data, regardless of the position of the communicating devices within the network. This also ensures the proper modularization of device implementations. Of course, the Java implementation of such a `VirtualConnection` object requires careful consideration (see section A.2.1 for further details), but the result is conceptually simple.

### 6.6.4 Multicast Connections

In more precise terms, the virtual connection described in the previous sections represents a *unicast* connection, i.e., data always flows from *one* output port toward *one* input port. In practice, *multicast* connection is also often required, meaning that, for example, data leaving one output port arrives to several input ports, conceptually copying the content of the media stream (see Figure 6-13). This copying should be oblivious to the source of the data. The behaviour of a virtual device should not depend on whether its generated data goes to one input device or more. A typical example for such setting is the well–known Internet Mbone service. A live video recording is done somewhere, this data is broadcast to the Internet, and listeners can “attach” themselves to this broadcast if they wish.

To fill this need, a subtype of the `VirtualConnection` object is defined in MSS, called `VirtualConnectionMulticast`:

[1] To be more precise: the buffer manager is automatically provided by the `java.io` classes!

```
public interface VirtualConnectionMulticast
   extends VirtualConnection, java.rmi.Remote {
   void attach(VirtualDevice device,  int portID)
      throws  ConfigurationMismatch, PortMismatch,
              ResourceNotAvailable;

   void detach(VirtualDevice device,  int portID)
      throws PortMismatch;
}
```

The interface of the operations is quite simple. Through the `attach` operation, the client can attach a new slave to the master port (remember that the `connect` operation of the `VirtualResource` object not only creates a connection between two ports, but also identifies a master and a slave port, see page 156). Setting up a new connection to the new slave involves the same steps and constraints as setting up the original connection. Obviously, one of the slaves can be detached from the master through the `detach` operation.

It should be emphasized, however, that behind this simple interface there may be a significant complexity when it comes to the realization of the multicast connection. The implementation of the kind of multicast connection depicted on Figure 6-13 is still relatively easy; some kind of "copying" engine has to be inserted between the output port of Device 1 and the stream. "Where" this copying should take place is not necessarily a simple issue, because one should try to minimize network traffic, but various optimization schemes are possible.

An especially difficult problem is encountered when the connection is "fan–in" (as opposed to a "fan–out" connection on the figure), i.e., when several sources can contribute to the same input port. Indeed, the integrity of the media data must be ensured, i.e., the media content should be "packaged" in a meaningful way, and these packages should be, essentially, atomic. Implementations of PREMO may choose not to implement a fully general "fan–in" multicast virtual connection. Instead, subtypes may be defined closer to the application domain, thereby making use of a more detailed knowledge about the content and the structure of the media data.

## 6.7 Groups

A more complicated network of virtual devices, with the corresponding virtual connections, can be very complex, and may include a large number of virtual resource objects. Controlling all these objects, such as acquiring their resources, stopping their progression, etc., may become a tedious task for the application program: it has to issue a large number of operation invocations repetitively to control the behaviour of the full network.

`Group` objects have been introduced by the MSS to ease this task. These objects (which are virtual resources themselves) offer a single entry points for the control of a number of other virtual resources. Groups, by default, do not have any special semantics (although subtypes of groups may, of course, introduce special behaviour). They act as some kind of "proxies" for other virtual resources.

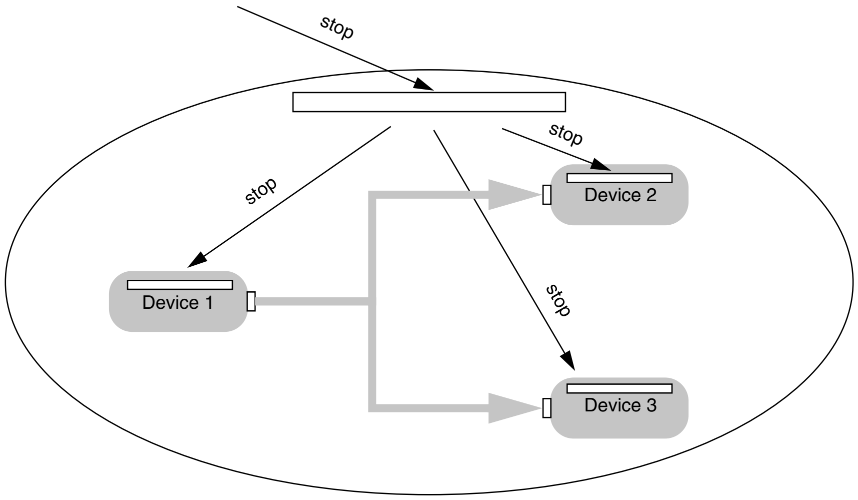


Figure 6-14 — Effect of a Group's Global Stream Control

Resources are added to a group by the operation:

```
void addResource(VirtualResource resource);
```

and are removed by the operation:

```
void removeResource(VirtualResource resource)
   throws ResourceNotAvailable;
```

Alternatively, the operation

```
void addResourceGraph(VirtualResource resource);
```

recursively adds to the group the `resource`, as well as all virtual connections and virtual devices which are connected, directly or indirectly, to `resource`[1]. Such a resource graph can be removed from the group through the operation:

```
void removeResourceGraph(VirtualResource resource)
   throws ResourceNotAvailable;
```

Finally, all resources, managed by the group, can be inquired by:

```
VirtualResource[] getResourceList();
```

Because they are virtual resources, groups also have global stream control objects, and they also implement operations such as `acquireResource`. However, in the case of groups, the only task these objects and operations have is to dispatch the "same" operation to the relevant methods of all virtual resources managed by the group. For example, if the `stop` operation of the group's global stream control is invoked, the effect is

[1] Note that the virtual device and the virtual connection types define the `getConnection` and the `getEndpointInfoList` operations, respectively, which help the reconstruction of the full graph.

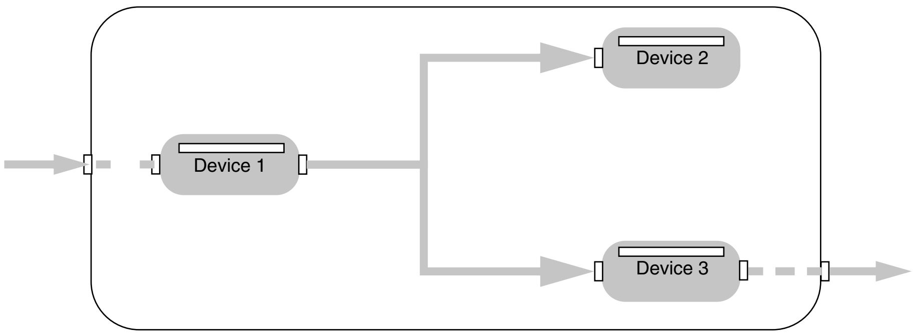


Figure 6-15 — Logical Device

to issue the `stop` operation on the global stream control objects of all constituent virtual resources (see Figure 6-14). Similarly, acquiring resources means to acquire the resource of all virtual resource objects managed by the group.

## 6.8 Logical Devices

Groups do not impose any restriction on the type of objects they manage. Also, in the inheritance hierarchy of virtual resources (see Figure 6-7), they represent an independent type both from virtual devices and virtual connections, i.e., they cannot “replace” any of those. Consequently, they are not usable, by themselves, to build up hierarchies of networks.

To construct hierarchies, logical devices can be used. The `LogicalDevice` type is a special subtype of `Group`, which is also a subtype of `VirtualDevice`. This means that a logical device can manage a collection of devices and connections, just like groups do, but they are also virtual devices themselves, which means that they can be included into a full multimedia network on their own right.

In order to communicate with other virtual devices, logical devices should also have (input or output) ports, and, to function properly, these ports should be related to the ports of the devices which are “managed” by the logical device. All this is done by the (only) extra operation defined for a logical device, namely:

```
int definePort(VirtualDevice refVirtualDevice, int portId)
   throws InvalidPort, InvalidDevice;
```

This operation refers to an existing port of one of the devices managed by the logical device, and instructs the logical device to create a new port which is “identified” with the argument port (the port ID for the new port is returned by the operation). One could also say that the internal port is “exported” beyond the boundaries of the logical device. This means that:

- The port configuration of the new port (as accessed and managed by a client through the virtual device interface of the logical device) is identical to the “real” port of the device contained by the logical device.

- All data flowing through the logical device's port is, conceptually, transferred unchanged to the "real" port.

Note that "internal" ports are not automatically exported; some ports, which are used for internal communication only, may stay hidden from the outside world. In other words, the logical device may also play the role of "information hiding" when constructing multimedia devices and therefore plays an important role in building up complex media networks.

# Chapter 7

# The Modelling, Rendering, and Interaction Component

## 7.1 Introduction

PREMO was initially envisaged as a new standard for computer graphics based on object-oriented technology. It was soon realised, however, that an equally significant problem in the design of "next generation" graphics applications was the need to integrate other media with graphics at a fundamental level, under the control of an API. Part 4 of PREMO, the Modelling, Rendering, and Interaction (MRI) Component is where this integration takes place in the standard. Thus, while the Multimedia Systems Services Component provides architectural support for viewing graphics processing in similar terms to other media processing applications, it does not directly address the content of the data used to describe the presentation. Instead, it defines streams and the concepts of processing resources that are independent of media content. In the MRI component, these facilities are used to define generic objects for modelling and rendering data, and basic facilities for supporting interaction. To support interoperability between devices for processing various media, the MRI component defines a hierarchy of abstract primitives for structuring multimedia presentations. Finally, it defines a specialised device for coordinating processing activities that operates on a heterogeneous multimedia presentation.

The interaction between the ideas in this chapter, and the concepts of MSS, is portrayed in Figure 7-1 which shows how a number of devices might be used in an video composition tool[1)]. Each of the oval shaped objects is a particular kind of MRI device and an MRI device is itself a subtype of the VirtualDevice object type from MSS. These devices extend the facilities defined in MSS and can be organised into a processing network, connected to each other by media streams. The main feature of an MRI device is the type of data that its streams carry: data that describes media derived from a collection of abstract primitives defined in the MRI component.

In the system, an audio modeller and a graphics modeller are being used to construct mono-media components of a presentation, which are stored in a database. A modeller in MRI is simply any device that can *produce* a stream of MRI primitives; a modeller may be a sophisticated interactive tool for constructing and editing presentation data, or nothing more than a device for accessing primitives stored in some external format. The

1) This and other examples in this chapter have been set up to illustrate aspects of the MRI component. They do not necessarily represent the most appropriate architecture for implementing such a system or facility.

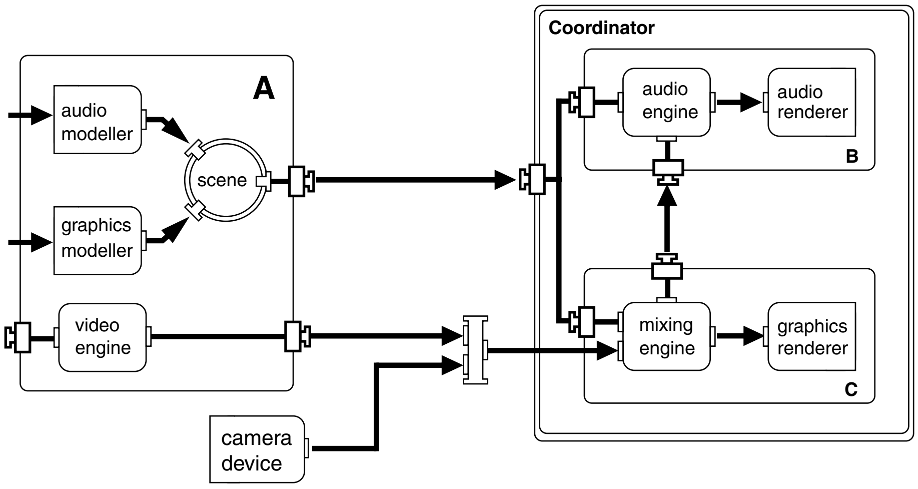


Figure 7-1 — A Configuration of MRI Devices

modellers, and the database that holds their output, are organised into an MSS logical device, along with an engine for processing video data. This device can be thought of as the "production" side of the composition tool, and might be implemented on a server.

The other 'side' of the system is concerned with mixing and rendering audio/visual data, and could be implemented on a client machine. At the top level, it consists of a single instance of an MRI device type called a Coordinator. A coordinator device receives a stream of presentation data, and is then responsible for allocating the component parts of that presentation to a collection of local resources that are able to process particular parts of the overall presentation. In the example, the local resources consist of an MSS logical device (B) that handles audio processing and rendering, and a second logical device (C) for video mixing and rendering. Both logical devices encapsulate a pair of MRI devices: an engine for processing specific kinds of media primitive, and a renderer that can take a stream of media primitives and convert it into some external representation, for example output on a display or through a sound system. The mixing engine in the example takes presentation data from the coordinator, and also has a video feed that can be switched by an MRI routing device between the video engine of the source, and some external camera device.

As the example suggests, the infrastructure provided by the MSS component is fundamental to the approach taken in MRI. Most of the devices used in the example can be defined as straightforward specialisations of the MSS `VirtualDevice` object type. The two exceptions are the `Scene` database and `Coordinator`, each of which will be considered in some detail in this chapter. What Figure 7-1 does not show, however, are the primitives carried by the media streams between the devices. These primitives interact with the definition of the MRI devices at a number of levels; while the devices process primitives, some of the functionality of the device is determined or influenced by the form that the primitives take.

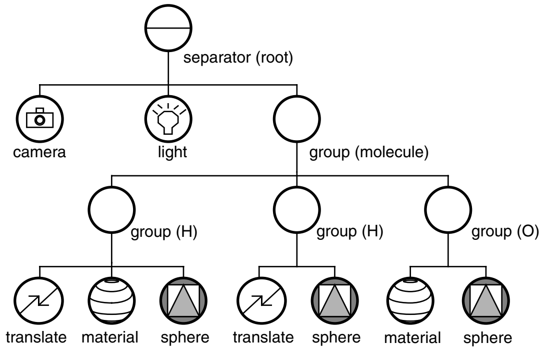


Figure 7-2 — A Scene Graph for a Water Molecule

## 7.2 Primitives

In computer graphics, a primitive is a basic building block for producing a picture. What exactly constitutes a ‘primitive’ depends of course on the graphics system, and the point within the ‘rendering pipeline’ being considered. For example;

- an API for rendering may allow the programmer to specify arbitrary polygons as geometric modelling primitives, but might internally convert all polygons into a collection of triangles to simplify processing;
- geometrical solids such as cylinders and pyramids might be considered as ‘primitives’ within one API, but be viewed as programmer-definable abstractions within another.

The tension in the design of rendering primitives can be characterised as between, on the one hand, a “minimal” set of orthogonal primitives that can be rendered efficiently by hardware, but which require programmer effort to organise them into larger structures, and on the other hand, a “comprehensive” set of primitives that provide a range of geometric building blocks, but for which a renderer has to do comparatively more complex processing within software.

Orthogonal to the “minimality” discussion is the question of what the notion of ‘primitive’ includes. Standards such as GKS and PHIGS distinguish between output (or rendering) primitives that define geometry to be displayed, and input primitives returned by an input device such as a pick or locator. Attributes of a model, such as colour, material properties (e.g. diffuse and specular reflection coefficients), line width, etc., are separate from primitives in GKS, PHIGS, and also OpenGL, as are geometric transformations and constraints. However, with the development of object-oriented approaches to rendering, as realised in systems such as Open Inventor and VRML, primitives, attributes and transformations have been abstracted into the concept of *nodes* within a *scene graph* that gives a declarative representation of the model to be rendered. For example, Figure 7-2 shows the scene graph for a simple model of a water molecule (using the node conventions found in [89]). While the meaning of each type of node clearly determines how nodes can be assembled to produce sensible results, the fact that all kinds of data – geometric detail, attributes, and transformations – are defined in similar terms means that there is great deal of flexibility in constructing and processing such a representation.

### 7.2.1 The Role of Primitives in PREMO

Whatever model is adopted, an API for computer graphics must support a set of primitives that is sufficiently complete to allow the programmer to produce output on a display. So what set of primitives should PREMO support? There are a number of possibilities:

- PREMO could define a new set of primitives. As PREMO aims to integrate synthetic graphics with other media, such a combination could be tailored to simplify the job of combining media. The problem with this approach is that it imposes a large overhead on a PREMO implementor. A powerful and efficient renderer, for example for PHIGS PLUS or Open Inventor, is a significant software development problem, and in the case of PREMO, is only one part of the requirements set out in the standard!
- PREMO could adopt an existing set of primitives. This would create two problems. First, which set of primitives should be selected? Both GKS and PHIGS are ISO Standards, and there is a case that a new standard should use established practice where possible. However it could equally well be argued that GL or Inventor are more widely used, or that VRML or Java3D are a new generation of graphics standards and will therefore define future practice. The second problem is that, apart from Java3D, these systems address only synthetic graphics. PREMO is intended to facilitate the design of multimedia presentations. Java3D does contain some provision for streaming media such as video and audio with synthetic graphics, but was not available at the time PREMO was designed; nor is it obvious even now whether the facilities it provides would address all of PREMO's needs.
- PREMO could define *abstract* primitives. Rather than the PREMO standard dictating the capabilities of a graphics renderer by fixing a set of primitives, a more radical and productive approach is for PREMO to interoperate with existing renderers. A PREMO presentation could thus be constructed from the set of primitives processed by the renderer available to a PREMO user. Indeed, a PREMO presentation could include primitives drawn from a collection of renderers, provided that we can ensure that a particular renderer only receives primitives it can deal with, or that renderers can deal with unrecognised primitives in a graceful way.

The third approach is particularly appealing, since it also allows a presentation to contain non-geometric primitives (for example, the description of an audio presentation), provided that there is a way for these primitives to be recognised by a suitable renderer. It also means the standard is fundamentally extensible; for example, by not committing to any one primitive set, new technologies such as haptic rendering could be incorporated without 'breaking' a pre-defined model.

It might seem from this discussion that there is actually no need for PREMO to define any kind of primitive set, it just needs to define an abstract type called "primitive" and allow primitives specific to a renderer to be identified as subtypes of this. This approach might be feasible if we consider a PREMO system as a single black box, however, the fundamental model of PREMO set out in the MSS component is that an application consists, in general, of a network of processing devices. Devices are objects, and therefore can be created to satisfy given requirements. Devices are connected by data streams, and if we are to interconnect devices, we must have some way of characterising the media data carried on a given stream, or produced / accepted at a given port of a device. Finally, if a PREMO presentation consists of a collection of heterogeneous media, it may be necessary to coordinate that presentation across multiple devices. To describe and manage the required coordination, it is necessary that there be some level of common organisation imposed on the underlying primitives. The PREMO primitive hierarchy meets these needs in two ways:

1. It defines an extensible collection of abstract object types for characterising the different *kinds* of primitive that a device within a PREMO network might encounter. These object types provide a small common 'vocabulary' for approximating the capabilities of various devices and the content of media streams.
2. One branch of the hierarchy introduces a minimal set of primitives that give a declarative model of a multimedia presentation. These are included in the standard since such facilities are not typically available within the primitive set provided by a particular renderer. A more detailed model of the overall organisation of multimedia content is also needed for the definition of the coordinator device defined in this Part (see Section 7.7).

To emphasise a point made previously, PREMO primitive hierarchy is not sufficient in itself to build a working presentation, but provides the abstract supertypes from which a set of concrete primitives could be derived by inheritance. The PREMO MRI component is not itself a rendering engine, but rather a framework for integrating media processing and rendering, performing a service for digital media somewhat like the service that a coordination framework, such as Linda [15] or Manifold [4], provide for concurrent processing.

### 7.2.2 The Hierarchy in Overview

Primitives are structures, that is, the object type *Primitive* inherits from *SimplePREMOObject*. At the top level, PREMO distinguishes between the seven kinds of primitive shown in Figure 7-3: *Captured*, *Form*, *Tracer*, *Modifier*, *Reference*, *Structured* and *Wrapper*. Each of these types of primitive is described in depth in a separate sub-section. The specification of the abstract type Primitive is simple:

```
    public class Primitive extends SimplePREMOObject {}
```

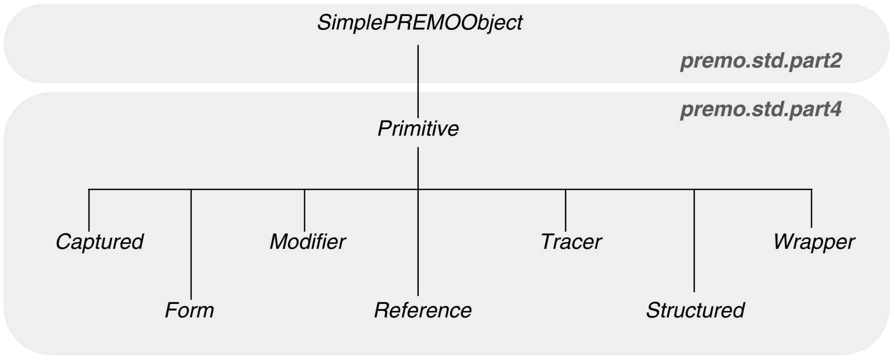


Figure 7-3 — PREMO Primitives: Top Level

## 7.2.3 Captured Primitives

`Captured` primitives form the primary interface between the notion of multimedia presentation defined by the MRI component, and the various standards for digital media encoding and transport that are in widespread use, for example ALAW, MIDI, MPEG, and ULAW to name a few. Computer graphics metafiles, including formats such as CGM or VRML, are also supported through the `Captured` object type. In general, a captured primitive is one for which some or all of the perceivable aspects of the primitive have been encoded in a format defined externally to PREMO. Rather than being synthesized, the presentation will be obtained 'ready-made' from some other component within a PREMO application.

As far as a PREMO application is concerned, the source of a captured primitive is immaterial; it may come from an external feed, e.g. a network or device interface, or be the product of some other processing device within the application itself. This flexibility is realised with the aid of the port and virtual device concepts of the MSS component. A `Captured` primitive thus consists of a reference to some virtual device that will produce the media data and a reference to a port of that device from which the data can be obtained. An application can determine detailed information about the format of the captured data by accessing the `Format` object attached to the port. The device and port references form the protected state information of the `Captured` object type:

```
public class Captured
    extends Primitive
{
    protected VirtualDevice srcDevice;
    protected int srcPort;
}
```

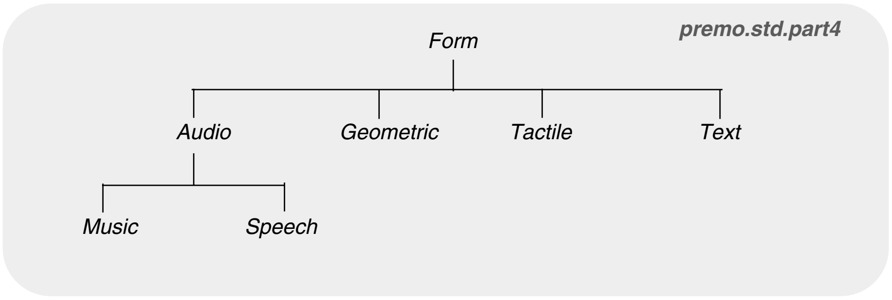


Figure 7-4 — PREMO Primitives: `Form` Hierarchy

### 7.2.4 Form Primitives

In contrast with captured primitives, where the presentation has been encoded in some internal format, the presentation described within a form primitive has to be somehow synthesised. For example, a graphical model might be described in terms of the geometry of polygonal surfaces. The actual appearance of the geometry as it is presented to an end user must then be synthesised from this geometric model, possibly taking into account other primitives such as visual modifiers (Section 7.2.6) that could alter the appearance of the geometry. The `Form` object type is abstract:

```
public class Form extends Primitive {}
```

In general, subtypes of `Form` can be said to describe structures in visual, audio, haptic or temporal space using the abstractions that characterise the space. For example, geometric primitives are usually described in terms of spatial coordinates or operations on simpler spatial structures (e.g. extrusions). The hierarchy of `Form` primitives is shown in Figure 7-4. Additional kinds of form primitives could be added in future to include other categories such as olfactory and taste (for example, olfactory rendering has potentially valuable applications in perfumery). The various specialisations of `Form` are described in the remainder of this section.

#### Audio Primitives

Audio rendering is now developing into a discipline in its own right, and involves a range of issues that go beyond the scope of this book or indeed PREMO. Some forms of audio presentation can be expressed as captured primitives, for example an ALAW file which carries a recorded signal. Other forms of audio primitive however represent an abstract encoding of audio information. Synthesized sound can be described using some abstract representation that operates in terms of the constituents of the sound. It can be divided into two general categories, music and speech. In the case of music, a typical example of a representation used for synthesized and sampled sound is MIDI. In PREMO, a music primitive contains information about the kind of instrument to be used in realising the sound, plus the data that represents the encoding of the music. Speech on the other hand consists of some textual representation of the words to be uttered. Other characteristics of speech, for example properties of the voice that should be used to render the text, are represented through a `VocalCharacteristic` object. This is an example of a PREMO modifier primitive, to be discussed in Section 7.2.6. The PREMO `Audio`, `Music` and `Speech` primitives are defined below.

```
public class Audio extends Form {}
public class Music extends Audio {
    public int instrument;
    public int score;
}
public class Speech extends Audio {
    public VocalCharacteristic voice;
    public String text;
}
```

By using `Aggregate` and `TimeComposite` objects to organise audio primitives into larger structures, more sophisticated sound characteristics can be described, for example by combining a number of audio primitives and acoustic effects into a score.

A MIDI hierarchy could be defined as an extension of the Audio primitive in order to integrate MIDI more directly into PREMO presentations, also allowing generation of MIDI scores with the use of PREMO primitives. This way, MIDI could be integrated in a PREMO presentation either as a captured primitive or a MIDI primitive.

#### Geometric Primitives

Geometric primitives have already been used in this chapter to explain some of the differences between PREMO and other standards, particularly in computer graphics. The intention of the PREMO designers is that the geometric primitives used by a renderer or needed by an application will be defined as subtypes of `Geometric`:

```
public class Geometric extends Form {}
```

For an object-oriented model of primitives, such as for example those underlying Open Inventor [89], the PREMO `Geometric` class would effectively become the top of the hierarchy.

### 7.2.5 Tactile Primitives

`Tactile` primitives describe parameters of touch-based interactions, for instance, temperature, thermal conductivity and hardness. They are present to support developments in haptic rendering, see for example [76].

```
public class Tactile extends Form {}
```

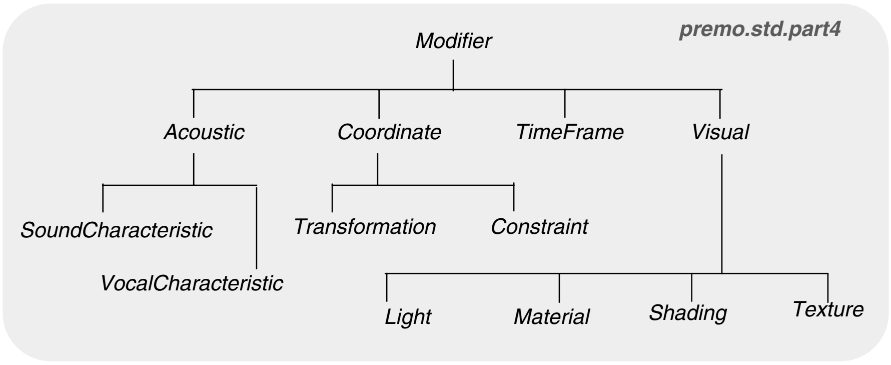


Figure 7-5 — PREMO Primitives: `Modifier` Hierarchy

**Text Primitives**

Text in computer graphics is typically used to label drawings, or, in the case of logos or other forms of advertising or labelling, as a geometric primitive in its own right. For a multimedia presentation, text has potentially wider roles, including for example subtitles on a video stream, or captions on figures or images. At an abstract level, a PREMO `Text` primitive simply contains a character string that is to be rendered on some display. No statement is made about properties of the text such as font, size, style or the direction in which it is drawn.

```
public class Text extends Form {
    StringBuffer characters;
}
```

Facilities for structured and/or formatted text can be realised by PREMO applications in at least two ways. Subtyping can be used to extend the text primitive or to indicate that the character string contains a particular kind of markup information. For properties such as font and character size that affect the presentation of the entire string, a new kind of modifier primitive (see Section 7.2.6) could be defined.

### 7.2.6 Modifier Primitives

Modifiers in PREMO are a generalisation of attributes within computer graphics. A `Modifier` primitive has no perceivable representation itself. Instead, modifiers affect the presentation of other primitives that are combined with the modifier through aggregation, which is described in Section 7.2.9.1. The modifier object types defined by PREMO are abstract, and therefore could have all been defined as direct subtypes of `Modifier`. The hierarchical organisation, shown in Figure 7-5, reflects the kind of effect each modifier produces, and the kind of primitives to which it can be applied.

```
public class Modifier extends Primitive {}
```

### Acoustic Modifiers

Acoustic modifiers alter the presentation of captured or synthesised sounds. Two kinds of acoustic modifier are represented by abstract subtypes of *Acoustic*.

- A *SoundCharacteristic* is a modifier that is defined in terms of the physical characteristics of a sound, for example its amplitude, envelope or other properties of its waveform. It could also represent the properties of a waveform which in turn modulates another, e.g., a sawtooth wave which modifies the amplitude (another modifier) of a sound.
- A *VocalCharacteristic* is a modifier that applies to synthetic speech, and affects the way in which the constituents of a given speech object are realized. Examples of possible vocal characteristics include sex, age, intonation and dialect.

The Java class definitions are again abstract. How information about acoustic modifiers is represented in a concrete class may depend on the representation chosen for sound within any `Audio` primitive, or within a captured audio stream.

```
public class SoundCharacteristic extends Acoustic {}
public class VocalCharacteristic extends Acoustic {}
```

### Structural Modifiers

Structural modifiers affect the interpretation of coordinate values representing, for example geometric structure or time, within some collection of primitives. Two kinds of structural modifiers have been identified, and are represented explicitly as subtypes:

- *Transformation* objects, which include, but are not limited to, the common affine and projective transformations. These include the 'standard' geometric operations such as translation, scaling, rotation and shearing.
- *Constraint* objects that serve to constrain the appearance of other geometric primitives. Constraints may be used to implement clipping, shielding, culling, level of detail objects or the definition of stencils.

Concepts of transformation and constraint can also be applied to non-geometric coordinates such as time, or colour. The structural modifier primitive and the given subtypes are abstract object types:

```
public class Structural extends Modifier {}
public class Transformation extends Structural {}
public class Constraint extends Structural {}
```

The structural modifications supported by a given application will depend in part on the *dimensionality* of the primitives used (see Section 7.3). Applications may also differ in whether they support an explicit representation, for example encoding a transformation as a matrix, or an implicit approach in which the parameters for the transformation are stored in the primitive and are then extracted by the device that processes the primitive.

## TimeFrame Modifiers

While the approach used by PREMO to represent multimedia presentations does not explicitly use any notion of a temporal primitives, the `TimeComposite` primitive and its subtypes that are described in Section 7.2.9.2 do refer to time units. In a distributed setting different clocks may be available, offering varying degrees of accuracy, and with different concepts of the current time. References to time within the description of a multimedia presentation must therefore be reconciled against some clock, and indeed different parts of a presentation may need to refer to distinct clocks. Rather than require that every reference to a time unit be accompanied by a reference to a clock, the MRI component provides a modifier, `TimeFrame`, that contains a reference to a clock. This can be combined with primitives using for example the `Aggregate` object type (see Section 7.2.9.1) to indicate that when processing the presentation, the media processor should use the clock given by the primitive.

```
public class TimeFrame extends Modifier {}
```

## Visual Modifiers

Visual modifiers in PREMO represent information that a renderer uses to affect the surface appearance of geometric primitives. In a graphics renderer, this encompasses material properties, settings such as colour and line width, and rendering parameters such as the shading model employed. Like other PREMO modifiers, these need to be combined with appropriate primitives through some form of aggregation. All visual modifiers are abstract object types.

```
public class Visual extends Modifier {}

public class Light extends Visual {}
public class Shading extends Visual {}
public class Texture extends Visual {}
public class Material extends Visual {}
```

1. `Light` is an abstract supertype for properties related to light. PREMO makes no commitment to any specific lighting model. It is up to an application to extend this type in a suitable way.
2. `Shading` is intended to represent information about the shading model or parameters that should be used to render some or all of a primitive structure.
3. `Texture` supports the definition and use of information representing surface detail, for example texture or bump maps.
4. `Material` is defined as a container for properties such as translucency and transparency.

### 7.2.7 Wrapper Primitives

Although it is not stated or required in the PREMO standard, primitives derived from many of the PREMO primitive object types can describe a multimedia presentation. An application may also want to treat input as a stream of primitives, and it is therefore useful to allow for this in the MRI component. However, there is even less consensus about input primitives than about output, in part because of the growing range and variety of output devices that are available. Rather than attempt to structure the space of values returned by input devices, PREMO provides a primitive called `Wrapper` that encapsulates an arbitrary non-object value. In the standard, the value carried by a `Wrapper` object is required to be of the non-object type *Value* that represents the union of the possible non-object types. As noted in Section 5.2.1, this union type is represented in the Java binding by the `Object` class:

```
public class Wrapper extends Primitive {
   Object content;
}
```

The data carried by a wrapper primitive will typically represent the measure obtained from some input device (e.g. the position of a locator as a `Coordinate` object, or the position of a valuator as a real value (represented by a Java `Double` object).

### 7.2.8 Tracer Primitives

The *Tracer* primitive is distinct because rather than defining it as a starting point for deriving application primitives, it has been defined as an object type to help the MRI component make use of the MSS facilities. A `Tracer` object contains a single variable that refers to an `Event` object (defined in Section 5.3.2.1):

```
public class Tracer extends Primitive {
   public Event trace;

   public Tracer(Event trace) {
      this.trace = trace;
   }
}
```

The `eventName` attribute of the event is set to `"TracerEvent"`, and the `eventSource` attribute is set to reference the `Tracer` primitive object in which the event is contained. This linking of the event to the containing `Tracer` object can be facilitated by the constructor.

The role of `Tracer` primitives will be explained when we describe MRI devices in Section 7.4. Briefly however, they allow such devices to determine the progress of primitives through a network built from the MSS components. We note in closing that when using `Tracer` primitives to monitor progress of data through a network, one must be aware that sending and processing a `Tracer` will itself require some finite time, which have to be taken into account when using these for synchronization. However, this duration is likely to be small in comparison with the time needed to process the data actually used to generate a presentation.

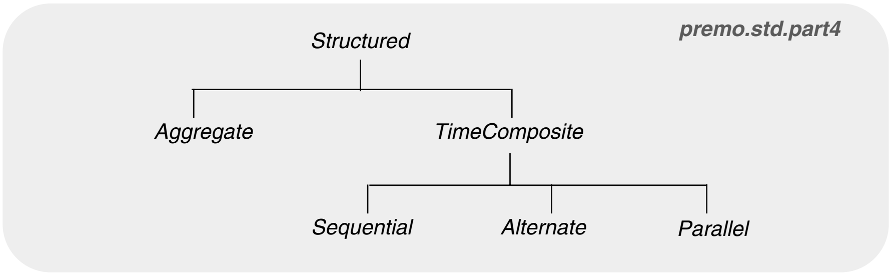


Figure 7-6 — PREMO Primitives: `Structured` Hierarchy

### 7.2.9 Structured Primitives

The object type Structured is defined in PREMO as the supertype of a group of object types, shown in Figure 7-6, that group together a number of simpler primitives. There are two main reasons why we want to group primitives, which are reflected in the two branches of the hierarchy below the supertype:

1. We may wish to group form or captured primitives with modifier primitives, to define or delimit the scope over which the modifier is applied, or simply to provide a level of hierarchy in the construction of the model.
2. We may wish to build an overall multimedia presentation by arranging primitives within some framework which indicates the order in which presentation should occur, and any temporal constraints that apply.

All structured primitives contain a collection of primitives, which in the Java model are stored as an array of `Primitive` objects. Each structured primitive also contains a `Name` object. For the present, it suffices to say that this provides a way of labelling the primitive. The role of `Name` is explained in more detail in Section 7.2.10.

```
public abstract class Structured extends Primitive {
   Primitive[] components;
   Name        label;
}
```

Note that any type of primitive can be a member of components, including further structured primitives. The standard does not explicitly rule out the creation of cyclic structures, so an implementation need not test for cycles when adding new primitives. The order in which primitives are stored as a component of a structure may be significant for some applications.

#### 7.2.9.1 Aggregate Primitives

Most graphics API's provide some means of organising primitives into structures that can be reused in multiple contexts and can be edited to reflect changes in the underlying model (e.g. structure editing in PHIGS [29]). Aggregates also have a role in delimiting

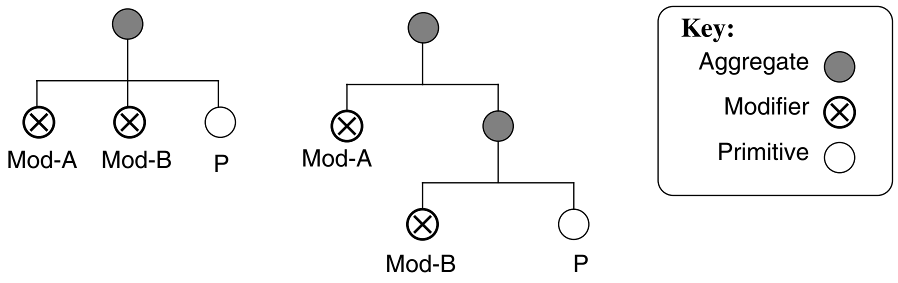


Figure 7-7 — Scope of Modifiers

the effect of transformations or attributes. For example, as shown in Figure 7-2, a 'group' node in Open Inventor can be used to combine geometric primitives (e.g. a sphere) with a property node that, for example, sets the material properties used to determine the appearance of the sphere when rendered.

Aggregates in PREMO allow a number of primitives to be combined into a structure without imposing an interpretation on the meaning of the structure. The `Aggregate` object type does not add features to the `Structured` object type, but is intended to act as a marker to show that this collection of primitives has been grouped together because the group conveys some meaning to a renderer.

```
public abstract class Aggregate extends Structured {}
```

`Aggregate` itself places no interpretation on its components. This will not be sufficient in general because there are different ways of interpreting a given collection of primitives. For example, the combination of a 3D point (a geometric primitive) and a MIDI file (a captured primitive) could either be rendered by displaying the point on some display and playing the contents of the file, or by interpreting the point as the location of the sound within the scene when 3D audio rendering is employed.

Although an `Aggregate` contains a sequence of primitives, PREMO does not prescribe the order in which modifiers are applied, and whether or not they are accumulative or override previous modifications. For example, in the two hierarchies shown in Figure 7-7, there is no requirement that Mod-A be carried out before or after Mod-B, and in the case of the second hierarchy, whether in fact Mod-B overrides any effect of Mod-A on the primitive P. Thus, the precise semantics of aggregation will depend on the actual renderer or device used to process the data. Different subtypes of `Aggregate` may be created to designate different effects. In particular, some models conflate aggregation with modification; a VRML [54] `Transform` node for example would be considered as both a modifier (a transformation) and an aggregate in the PREMO framework, because the transformation specified by the node is applied to each of its children. An implementation of VRML that is built on the PREMO primitives could implement a `Transform` class by subtyping from both `Aggregate` and `Transformation`.

### 7.2.9.2 TimeComposite

Although animation is a fundamental area of application for computer graphics, most graphics renderers operate on primitives that make no reference to time. Instead time is handled by separate language or system dependent mechanisms. This implicit approach is not satisfactory when we extend the presentation model towards dealing with multimedia data in general. Time and temporal extent are fundamental to multimedia presentation and, in general, a multimedia presentation will consist of media data that need to be synchronized. Time affects multimedia presentation at a number of levels. For example, the time at which a primitive is presented may be adjusted dynamically to satisfy quality of service requirements, while synchronization requirements might be realized by placing synchronization elements in TimeSynchronizable components of a PREMO system. Time thus plays a specialised role within multimedia which is not reflected, for example, in geometric coordinate spaces. Object types that define and manipulate temporal aspects of a presentation must therefore have a standard and efficient means of representing and accessing this information.

Object types for working with time at a low level have been described in Chapter 5. At the level of the MRI component, we are concerned with how a multimedia presentation is "laid out" in time. This information is represented in the `TimeComposite` object type and three specific subtypes. `TimeComposite` is itself a subtype of `Structured`, and therefore contains component primitives that are parts of some presentation. Exactly how the presentation of such components should be coordinated is defined in the subtypes of `TimeComposite`. PREMO recognises sequential, parallel and alternative composition, the meaning of which are defined later. All `TimeComposite` objects have certain characteristics, present in the superclass as shown below:

```
public class TimeComposite extends Structured {
   long     min, max;
   long     startTime, endTime;
   Callback monitor;
}
```

All `TimeComposite` objects contain a reference to an event handler (the `monitor` variable) that can be used in a PREMO system to keep track of when significant points in the structure of a presentation have been reached during rendering or other forms of processing. Subtypes of `TimeComposite` define particular points in their temporal layout at which the monitor will be notified of an event when they are being processed by a suitable kind of device. The other components of the `TimeComposite` class are as follows (please refer also to Figure 7-8):

1. `min` and `max` are time values (i.e. numbers of ticks) that define a duration within which the contents of the `TimeComposite` object should be presented / processed. The clock used to measure this interval is not specified by the primitive; it may be given by a `TimeFrame` modifier somewhere in the specification, or it may depend on the context in which the primitive is processed. The standard allows for an infinite interval, in which case an implementation is free to use as much time as necessary to process the components. This possibility has not been included in the Java binding described here, as it would complicate the definition of time without adding

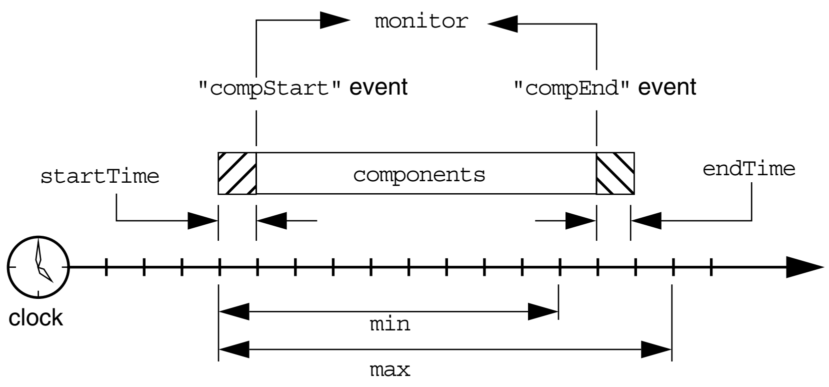


Figure 7-8 — Attributes of a `TimeComposite` Primitive

much illumination. The simplest approach would be to create a Java class which, like the built-in `java.lang.Double` class, defines constants that may be used to denote plus or minus infinity. In cases where the interval is finite, processing devices will need to deploy a suitable strategy to ensure that presentation takes place within the specified bounds. To keep activity within the allowed maximum interval, a device may need to use the temporal flexibility allowed for within the component primitives, or may have to degrade the quality of service that it provides. In the case that the processing cannot be performed at a required quality of service within the time allowed, the device may take some application-dependent action, for example raising an exception or aborting the task.

2. `startTime` is an offset that allows some latency between the point at which a processor receives a structured primitive and when it begins to operate on the first of its components. An event should be generated once the `startTime` offset has elapsed, and be sent to the event handler designated by `monitor`. The event name is set to `"compStart"`, and the data consists of a single key–value pair that associates the key `"TimeComposite"` with a reference to the `TimeComposite` object being processed. The event source is set to the processing device.

3. `endTime` is an offset between the time that the last component of a `TimeComposite` is processed, and the time at which processing of the `TimeComposite` itself is deemed to be complete. It allows a processing device to carry out any finalisation or housekeeping before the end of the period in which the composite media should be processed has elapsed. As with the `startTime` offset, an event is generated, in this case on completion of the presentation of the components, before the `endTime` delay begins. The event is similar to that used for `startTime`, except that the event name field takes the value `"compEnd"`.

Each of the three subtypes of `TimeComposite` inherit these features but process components in a different manner.

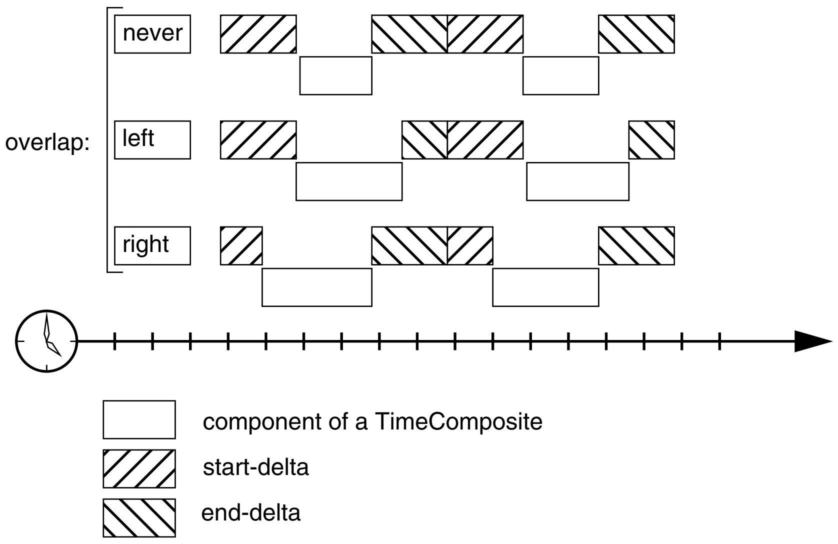


Figure 7-9 — Organisation of a `Sequential` TimeComposite Primitive

#### *7.2.9.2.1 Sequential*

In a sequential presentation, each component of the structured primitive is presented in turn, in the order in which they appear in the `component` array. A typical example of this kind of composition would be a film, with a title, content, and then credits, or a business presentation consisting of a sequence of slides or segments of other media. The overall structure of the sequence is given by the state inherited from TimeComposite; the PREMO `Sequential` object type adds attributes that specify temporal "padding" between the constituents, and describes how, if at all, the contents of component primitives might be allowed to intrude into this padding or indeed be truncated.

```
public class Sequential extends TimeComposite {
   long        startDelta, endDelta;
   OverlapType overlap;
}
```

`startDelta` and `endDelta` both represent intervals of time; `overlap` is an enumerated type that effectively has three possible values: `left`, `right` and `never`. Its implementation follows the scheme set out in Section 5.2.2, and will not be given in detail here. The meaning of the attributes is explained below, with reference to Figure 7-9.

1. `startDelta` defines an offset between the time that a device selects the next component of the primitive to process, and the time at which processing of that component begins. At the end of `startDelta` an event is sent to the event handler referenced by `monitor` (inherited from `TimeComposite`). The event name is `"seqStart"`, the source is the processing device, and the data consists of two key–

value pairs: the key `"sequential"` is bound to a reference to the `Sequential` primitive, and the key `"position"` is bound to the index of that component within the sequence of components.

2. `endDelta` is an offset between the time a device completes processing the data of a component, and the time at which it selects the next component. An event is generated once the component data has been processed, before the start of the delay. The structure of this event is similar to that used for `startDelta`, except that the event name takes the value `"seqEnd"`.
3. The values taken by the `overlap` attribute define how, if needed, the delays on either side of a `Sequential` primitive component can be used when processing a component.
   - 3.1. The value `left` allows a processing device to reduce the `endDelta` delay allocated to a component to allow additional time to complete processing of the data associated with that component. If by using all of the `endDelta` buffer there is still insufficient time to process the data as required, some application-specific action such as raising an exception, or an event, may be taken.
   - 3.2. The value `right` allows the startDelta buffer to be reduced to provide additional processing time for the component data.
   - 3.3. An overlap value of `never` requires a device to respect the `startDelta` and `endDelta` offsets for all components.

   It may seem that a processing device requires some form of oracle to determine whether or not to intrude into start or end deltas when it is allowed. If a device is processing a continuous data stream directly, it may not be able to use overlap information. The main role of this information, however, is to inform devices such as the `Coordinator` (see Section 7.7) that may schedule the components of a structured media primitive before carrying out the processing according to that schedule.

##### *7.2.9.2.2 Parallel*

Parallel presentation of data is one of the hallmarks of multimedia, for example video and audio, or audio and animation (synthetic graphics). Indeed, separate elements of an animation can be seen as parallel 'tracks' of synthetic graphics. The presentation of all components of a `Parallel` primitive occur concurrently. Even in a single-processor system where presentation is by necessity realised through time slicing, the intention is usually that the media streams be perceived as truly parallel.

```
public class Parallel extends TimeComposite {
   boolean startSync, endSync;
}
```

The `Parallel` object type introduces two boolean-valued attributes that describe how processing of the starts and ends of the component primitives should be aligned.

1. When `startSync` is `true` it indicates that the presentation of all components of the `Parallel` primitive must start at the same time. When it is `false`, a processing

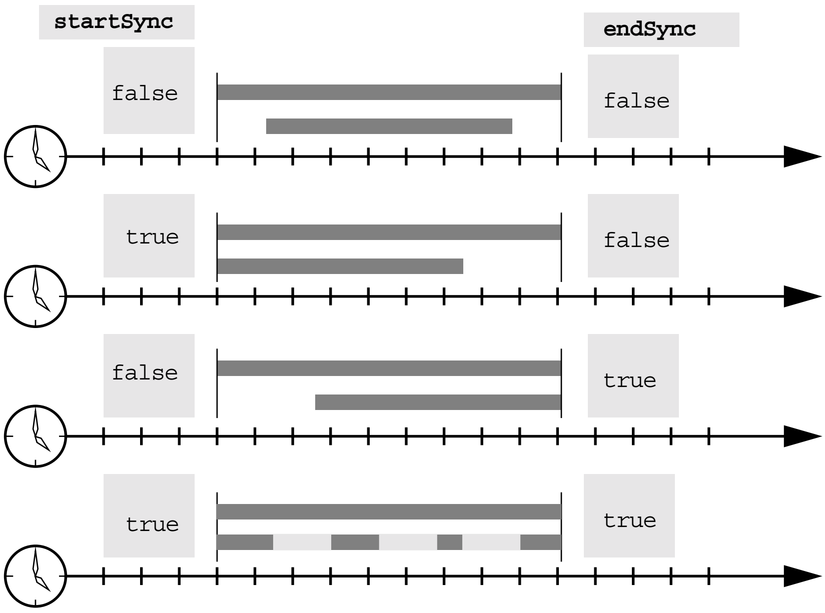


Figure 7-10 — Organisation of a `Parallel` TimeComposite

device can impose a delay before commencing to process some of the components. Such a delay may be helpful if `startSync` is `false` but `endSync` is `true`.

2. When `endSync` is true, it indicates that the presentation of all components of the primitive should end at the same time. Otherwise, processing of components can end at arbitrary times.

The interplay between `startSync` and `endSync` is shown in Figure 7-10. Note that here, as with `Sequential`, devices that carry out a scheduling step before processing media content will benefit the most from this information. In the case that both `start-Sync` and `endSync` are true, a device may have the freedom to realise this requirement by altering how it processes the media content. For example, if a component is itself a `TimeComposite` primitive, the various delays or temporal buffers within the primitive might allow the presentation to be stretched or compressed to fit the required interval. For other media, sampling or interpolation of content may be possible. However, there is no guarantee that a device will be able to satisfy such a constraint, and in all cases the `startSync` and `endSync` flags are understood not as hard constraints, but as requests for a device to make the "best possible" effort to align the processing.

The temporal extent of a `Parallel` primitive is taken to be the maximum of the extents of its components. When a component of a `Parallel` primitive is presented, and one of `startSync` and `endSync` is `false`, the "gap" between the start/end of the overall presentation, and the start/end of that primitive's presentation, will be filled in an application-specific way. For example, the unused time allocated to an audio stream will usu-

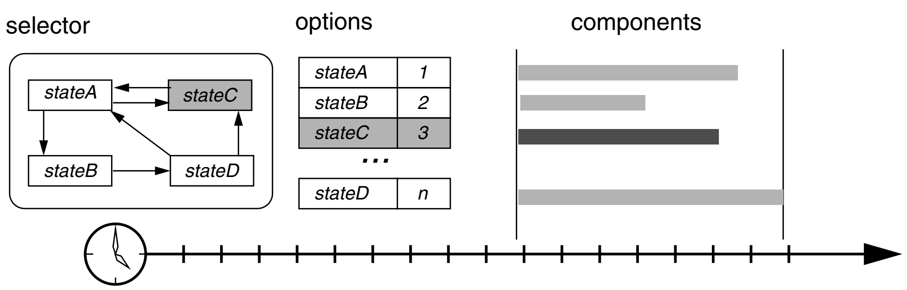


Figure 7-11 — Organisation of an `Alternate` TimeComposite Primitive

ally be filled with no sound at all. For a video stream, the video presentation might be "switched off" to leave a blank display area, or the final frame of the video might be continued until the remainder of the `Parallel` primitive has been presented.

### *7.2.9.2.3 Alternate*

Unlike the `Parallel` and `Sequential` primitives, in which each component is processed and presented, only one of the components of an `Alternate` primitive is processed by a given device. The component is selected based on the state of a `Controller` object, referenced from within the primitive (the `Controller` object type is described in Section 5.4.2). The state is checked by a device when it begins to process the primitive. Clearly care is needed when changing the state if the primitive is being passed through a network of devices. There are a number of well known applications or techniques that can be implemented using a facility such as that provided by `Alternate`, including:

- interactive media, where user input determines the presentation content
- alternative presentations, for example providing a text option to use in place of an image if a device is incapable of displaying that image
- level-of-detail objects, where a representation of, for example, a graphical object is selected based on the distance between the object and the viewer – this generalises to other media such as audio.

The definition of the `Alternate` type is as follows:

```
public class Alternate extends TimeComposite {
   Controller          selector;
   AlternateSequence[]  options;
}
```

The decision over which component primitive to process is determined by the state of a `Controller` object referenced by `selector`. Because the states of a `Controller` are identified by strings, there is a need to define a mapping from a string to an index of a primitive within the `component` sequence inherited from `TimeComposite`. This mapping is defined in the Java binding as an array of `AlternateSequence` objects, each of which contains a string, and an integer:

```
class AlternateSequence {
   String   state;
   int      stateValue;
};
```

An `AlternateSequence` object with `state` s and `stateValue` i within the `options` array indicates that, when `selector` is in state ‘s’, the primitive `components[i]` should be used. This arrangement is illustrated in Figure 7-11. The `AlternateSequence` class is local to `Alternate`.

### 7.2.10 Reference Primitives

A fundamental technique in computer graphics is to define a modelling primitive in some local coordinate system, and then re-use this primitive, subject to transformations, in various parts of a scene. This approach is supported by mechanisms such as structure invocation in PHIGS [29], multiple references to scene objects in Open Inventor [89], or the DEF/USE facilities in VRML [54]. This capability is reflected in the PREMO model through `Reference` primitives. These define an implicit link to some node in a structured presentation through a single variable which references a `Name` object. `Name` objects can be associated with any subtype of `Structured`, and a `Reference` primitive is thus interpreted as a link to the `Structured` node that contains a matching `Name`.

```
public class Reference extends Primitive {
   Name label;
}
```

As PREMO is intended to interoperate with different sets of modelling primitives, it provides a generic mechanism for dealing with naming, that can be interpreted or implemented in a number of different ways. The label of a `Name` object consists of a sequence (i.e. array) of strings, and the object type provides a method, equal, that is required to return boolean value indicating whether or not the receiver (in Java terms) and the parameter should be considered to represent the same name.

```
public abstract class Name extends SimplePREMOObject {
    String[] tag;
    public boolean equal(Name otherName);
}
```

By allowing for sequences of strings, the `Name` object type can be used to implement a directory-like naming scheme (where the order of names is important), or a mechanism like the GKS name set [48]. What PREMO does not define is how, given a `Name` primitive and some hierarchy of primitives built from `Structure` and its subtypes, the search for a matching name takes place (e.g. breadth first, depth first). This will depend on the renderer or media processor operating on the primitives.

## 7.3 Coordinate Spaces

PREMO primitives are abstract, so they do not contain explicit reference to values or structures that characterise media data, for example vertices, normals or colour for geometric data, volume or frequency for audio data, etc. However, it is still necessary to have some way of characterising "low level" media data within the MRI component. The reason is that different devices that process MRI data may wish to define properties in terms of the primitives that they can accept or produce. For example, a geometric modeller may produce output in either 2, 3 or 4 dimensional space. If the output of this modeller is to be passed to a renderer, it is important that the renderer be able to accept primitives defined over the appropriate space. Similarly, it is convenient to have a shared notion of how points in absolute coordinates are referenced (as opposed to the relative durations or intervals used in the `TimeComposite` primitives).

## 7.3.1 Coordinate

To support property-based negotiation between devices, and the description of properties, such as absolute time, that are defined within some coordinate framework, PREMO introduces the concept of a `Coordinate` object:

```
public abstract class Coordinate extends SimplePREMOObject {
   static int dimensionality;
   public int getDimensionality();
   abstract public double[] getRange(int dimension);
   abstract public void setComponent(int dimension, double value);
   abstract public double getComponent(int dimension);
}
```

A `Coordinate` object represents a point in an n-dimensional coordinate space. The dimensionality of the space is given by a read-only attribute of the object type, and should be fixed by subtyping from `Coordinate`. In the standard, `Coordinate` is defined as a generic type; the values that appear in the various dimensions are drawn from a generic parameter that can be instantiated to some arbitrary (ordinal) type. Rather than complicate the Java binding by incorporating this level of generality, we have assumed that each dimension of the `Coordinate` object type is a `double` value. Three operations can be used to get / set attributes associated with each component (i.e. a given dimension) of a `Coordinate`:

1. The range of values that can be stored in a dimension can be queried. The range is represented by a minimum and maximum value, stored as a 2-element array.
2. The value of a component can be set.
3. The value of a component can be enquired.

The standard does not define the result of attempting to access or set information about a component outside the dimensionality of a given primitive, nor does it state what should happen if `setComponent` is called with a value outside of the range for that component. The reason for this is that for applications that use `Coordinate` objects for large-scale data sets, the overhead of checking that values are within a specified range may be prohibitive. Therefore, PREMO provides an interface that allows such checking to be carried out, but leaves it to an implementor to decide whether the cost of doing so can be justified. Of course, an implementation could define a subtype of `Coordinate` that does in fact implement the check and takes some appropriate action if a dimension or component value falls out of bounds.

### 7.3.2 TimeLocation

Two specific kinds of coordinate space are defined in MRI component as subtypes of Coordinate. The first, `TimeLocation`, is a 1-dimensional space that represents locations in time by a number of ticks relative to some clock. The clock is not specified as part of the `TimeLocation` object because the storage overhead would be prohibitive for some applications. Those that wish to keep an explicit link to the reference clock can subtype `TimeLocation` and include a reference to a `Clock` object.

```
abstract public class TimeLocation extends Coordinate {
    static {
        dimensionality = 1;
    }
}
```

The `TimeLocation` class simply fixes dimensionality to be 1.

### 7.3.3 Colour

To provide devices with a minimal capability to negotiate colour models, an object type called `Colour` is defined which represents a point (i.e. a colour) within a specified 3-component colour model. The colour model is defined as a read-only attribute of a colour object, and can take on one of four values: `RGB`, `CIELUV`, `HSV`, and `HLS`. The fact this is a minimal framework for describing colour is reflected in the lack of support for an alpha channel.

```
abstract public class Colour extends Coordinate {
   static {
      dimensionality = 3;
   }
   public String getColourModel();
   public void setColourModel(String newColourModel);
}
```

A number of constants are defined in the standard to provide symbolic names for the components of each colour model, for example `ColourRGBR`, `ColourRGBG`, `ColourRGBB` are numerical constants that map the three dimensions of RGB colour space into dimensions of a PREMO `Colour` object.

## 7.4 Devices for Modelling, Rendering, and Interaction

The MSS component defines the concept of a *virtual device* and provides a collection of object types whose services support the implementation of a network of devices. However MSS makes no commitment about the content carried on the streams that link devices. As explained in the introduction to this chapter, the MRI component uses the `VirtualDevice` concept as a basis for specialised processing devices, as shown in Figure 7-1. Having described the PREMO model of multimedia content, we can now introduce the kinds of devices that can operate on that content.

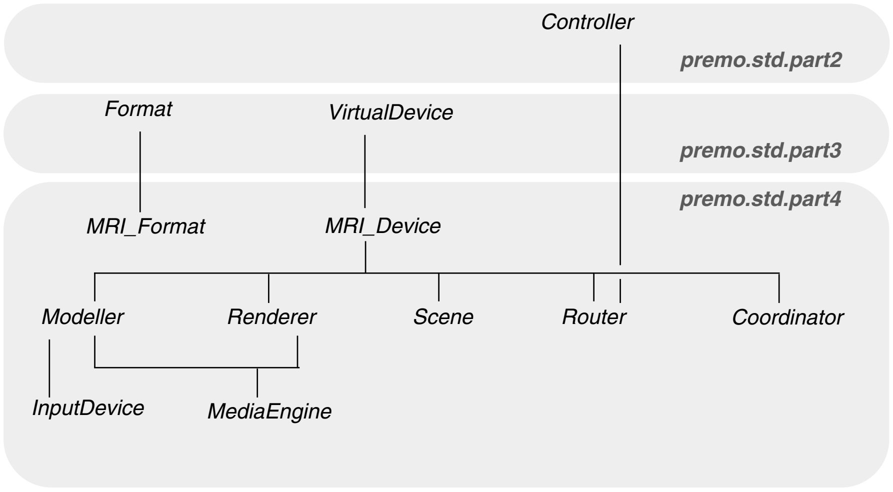


Figure 7-12 — `MRI_Device` Hierarchy and `MRI_Format`

### 7.4.1 MRI_Format

The devices defined in the MRI component for processing media data form a hierarchy that extends the `VirtualDevice` type of MSS. As shown in Figure 7-12, all such devices in the MRI component extend the type `MRI_Device`. Because all devices defined in the MRI component are virtual devices, they can be integrated into the kind of processing network supported by MSS, and where sensible, can interoperate with media-specific devices. The characteristic of an MRI device that distinguishes it from other kinds of virtual devices is that it has at least one port at which it can accept or produce a stream of primitives that are derived from the `Primitive` object type described earlier in this chapter.

Recall from Chapter 6 that each format that can be accepted or produced by a `VirtualDevice` at a given port is characterised by a `Format` object. Although the MRI component does not prescribe how primitives are transported across media streams[1], it does define an object type, `MRI_Format`, that represents a specialisation of the MSS `Format` object type for representing information about the transport of MRI data. Two properties are associated with `MRI_Format` objects and can be used, as with any other property, to negotiate the setup of an MSS network.

[1] One way of handling this in the case of the Java binding is to use Java serialization and to use `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` to write and read the primitives.

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `DimensionsK` | `DimInfo[]` | no | The dimensionality of the space in which a given kind of primitive will be defined. |
| `PrimitivesK` | `String[]` | yes | The kinds of primitive that can be accepted by the port. |

The data type `DimInfo` is used to represent the combination of a primitive type (represented in Java by a `Class` object) with an integer representing the dimensionality of that kind of primitive. It is implemented in Java as a simple class. Note that an implementation should ensure that the `Class` object stored as `primitiveType` corresponds to `Primitive` or one of its subtypes.

```
public final class DimInfo implements java.io.Serializable {
   public Class   primitiveType;
   public int     dimensionality;
}
```

Each of the properties has an associated capability:

| Key | Type | Value |
|---|---|---|
| `DimensionsCK` | `DimInfo[]` | `Not defined by PREMO` |
| `PrimitivesCK` | `String[]` | `Not defined by PREMO` |

For a specific device, these capabilities define the combination of dimensionality and kinds of primitive that can be accepted by the device at a particular port.

### 7.4.2 Efficiency Measures

In a rich multimedia environment, more than one device may be available to process a given set of primitives. Other constraints aside, one way of selecting among devices is to consider the efficiency with which they can process different kinds of primitive. Rather than attempt to prescribe some particular metric (e.g. "polygons per second"), an MRI device is equipped with an `EfficiencyMeasure` object. This object has two roles:

1. It provides a location where information about the efficiency of the device for some given characteristic can be stored. PREMO does describe how the performance of a device is to be encoded. `EfficiencyMeasure` objects should be associated with particular property keys of a device.

2. It defines an operation that accepts an `EfficiencyMeasure` (from another device) and produces one of four results: `worseThan`, `equivalentTo`, `betterThan`, `notComparable`. The result indicates how the receiver's device should be ranked relative to the argument's device in terms of the criteria on which the `EfficiencyMeasure` is defined.

The definition of the interface is given below:

```
public interface EfficiencyMeasure {
   public ComparisonRes compare(MRI_Device alternative);
}
```

For completeness, here is the enumerated type:

```
public final class ComparisonRes
   extends premo.impl.utils.PREMOEnumeration {
      public static ComparisonRes worseThan;
      public static ComparisonRes equivalentTo;
      public static ComparisonRes betterThan;
      public static ComparisonRes notComparable;
}
```

### 7.4.3 MRI Device

The main difference between the interface of `VirtualDevice` and that of `MRI_Device` is the presence of ports that accept `MRI_Format`, it should not be surprising to find that this is the focus of the main behavioural difference between the two types. When a port of an `MRI_Device` is configured to use `MRI_Format` (either for input or output), the device is required to monitor the primitives passing through the port. When a `Tracer` primitive is encountered, the event associated with the `Tracer` is dispatched to the event handler associated with that port.

This behaviour is motivated by a need to track the progress of primitives through a processing network. For media such as video that is transmitted as frames of an absolute size, the `StreamControl` object type associated with a device port can provide a measure of progress through the media. Although PREMO does not define the structure of the format used to communicate primitives across streams, in practice we expect most approaches to work by serializing the primitive and then using some form of object stream. In this case, it is more difficult to measure the progress of the transfer. Furthermore, the stream facilities provided by the MSS component provide no built-in means for acknowledging receipt, so the sender must make its own arrangements to be informed when media data has successfully arrived. It can do this by registering itself with the event handler of the port to which it is sending MRI primitives, and then placing a `Tracer` primitive on the stream to that port immediately after media data. MSS streams are required to preserve the order of content, so when the sender is notified of the arrival of the `Tracer` primitive, it means that the preceding primitive has also arrived.

`MRI_Device` places no requirements on whether `MRI_Format` is supported on input ports, output ports, or both. The constraints on what type of port needs to be `MRI_Format` capable are encapsulated within specialised devices, which are considered next.

### 7.4.4 Modeller

`Modeller` is an `MRI_Device` subtype for which at least one output port is capable of supporting `MRI_Format`. The PREMO concept of modeller is a generalisation of that found in computer graphics, where a modeller is a software package or interface that allows the creation of primitives, typically via specialised operations or services provided by the software. In PREMO terms, a modeller is a device that accepts input in some form undefined within the MRI component, but which can produce MRI primitives as output. It thus serves as a bridge from the application world into MRI devices and processing. Because the differences between MRI_Device and Modeller are in the resources that an object of the type provides, the Java interface for the Modeller is trivial:

```
public interface Modeller extends MRI_Device, java.rmi.Remote {};
```

One property is defined for this object type.

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `EfficiencyOutK` | `EffInfo[]` | yes | An `EfficiencyMeasure` for each kind of primitive the device is able to produce. |

`EfficiencyOutK` gives, for each primitive that the modeller can produce, an `EfficiencyMeasure` object that can be used to compare this modeller against another. A Java class, `EffInfo`, provides a data type for coupling a primitive type with an `EfficiencyMeasure` object.

```
public final class EffInfo implements java.io.Serializable {
   public Class           primitiveType;
   public EffiencyMeasure  efficiency;
}
```

`EfficiencyOutK` is clearly most useful when the primitives are being generated computationally, rather than interactively with some sort of editor.

## 7.4.5 Renderer

A `Renderer` is the dual of a `Modeller`; it is an MRI device that provides at least one *input* port that accepts `MRI_Format` data.

```
public interface Renderer extends MRI_Device, java.rmi.Remote {};
```

The PREMO `Renderer` object generalises the concept of renderer found in computer graphics. A PREMO `Renderer` is a device that takes MRI primitives, and processes them to produce some output that is beyond the scope of the MRI component. It may, for example, generate a file containing an image stored in some externally specified format, or may present the processed media data directly to an end-user.

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `EfficiencyInK` | `EffInfo[]` | yes | An `EfficiencyMeasure` for each kind of primitive the device is able to consume. |

### 7.4.6 MediaEngine

An object that can both accept and produce streams of MRI primitives is called a `MediaEngine` in PREMO, and is a subtype of both `Modeller` and `Renderer`. This is reflected in the Java binding by having the `MediaEngine` interface extend both `Modeller` and `Renderer`.

```
public interface MediaEngine
   extends Modeller, Renderer, java.rmi.Remote {}
```

`MediaEngine` thus inherits both the `EfficiencyInK` and `EfficiencyOutK` properties that characterise its ability to produce and consume primitives. However, these activities are usually coupled: the primitives produced by a `MediaEngine` will typically be derived from those that it accepts. A potentially more useful way of characterising the processing capabilities of a `MediaEngine` is in terms of its ability to transform or transmute one kind of primitive into another. This is specified by the `TransmutationK` property.

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `TransmutationK` | `MapEff[]` | yes | A measure of the efficiency with which one kind of primitive can be mapped into another. |

The property value consists of an array of objects, each consisting of a mapping defined by the types of the input and output primitives, and a reference to an `EfficiencyMeasure` object. Two Java classes are used to define this data type, since Part 4 of the PREMO standard defines `PrimMap` as a non-object data type (it potentially has use beyond this particular role):

```
public class PrimMap implements java.io.Serializable {
   public Class   input;
   public Class   output;
}

public final class MapEff implements java.io.Serializable {
   public PrimMap            mapping;
   public EfficiencyMeasure  efficiency;
}
```

An instance of `MapEff` in the `TransmutationK` property of a `MediaEngine` thus represents the efficiency with which that `MediaEngine` can map primitives that are instances of the input object type into instances of the output object type.

## 7.5 Input Devices, and Routing

A multimedia application developed using PREMO may require input at a number of levels. The simplest case is when input is required by a specific device, and when that input can be taken from an input device (e.g. keyboard, mouse) associated with the end user's machine. That is, as far as PREMO is concerned, the existence of input is hidden within the definition of the device itself. This however may not always be the case. For example, a speech recogniser operating a remote device may be best thought of as a device in its own right that produces a stream of data for another processing device. In this respect, an input device can be seen as a specialised form of `Modeller`, taking data from the "outside" world (in this case from an end user) and transforming it into a form that can be passed around and processed by arbitrary MRI devices.

### 7.5.1 InputDevice

`InputDevice` is a subtype of `Modeller` that is specialised to reflect three of the most widely used methods for processing input, namely request, sampled, and event-driven modes [29]. The operation of these modes is best understood by considering each input device to be capable of providing a *measure* and a *trigger*. The measure of a device is its state at some point in time. The trigger is a transition, initiated by the end user, that defines a point in time. The interplay between these two concepts is fundamental to the different modes:

- In *request* mode, a client that requires input invokes a synchronous operation on a device, and is thus suspended until the device yields a result. The device in turn may need to prompt the end-user to indicate that a particular type of input is required. The value returned by the input device is the measure at some point in time after the user has been prompted for input. This point in time is usually specified by the device trigger.
- Input in sampled mode takes place when the client inspects the "current" state of the device's measure. In a distributed context, the term "current" clearly has to be understood with transmission delays in mind. The sample returned is by definition the measure of the device; the trigger plays no real role in this mode.
- Event driven input takes place when a device trigger is converted into an event, which is then used to notify interested parties that input has become available. The measure of the device at the time the trigger occurred can be carried as part of the event data. Event mode has become particularly popular for a range of input devices through the widespread use of callback routines.

PREMO supports all three modes through the interface of `InputDevice`:

```
public interface InputDevice
   extends Modeller, java.rmi.Remote {
   public Primitive request();
}
```

Input in request mode is achieved by invoking the (synchronous) request operation. Sampling is supported by the primitive stream, while event mode is implemented via the event handler associated with the device. A device might generate several different events, which a client can inspect via the following property:

| Key | Type | Read–only? | Description |
|---|---|---|---|
| `EventNamesK` | `String[]` | yes | Event names that this device can raise. |

The measure of an event is returned via `eventData` using suitable object types from the MRI component. Geometric information, e.g. a location on the display, might be represented directly as a `Coordinate` object. More abstract forms of input, for example a valuator position, can be encoded using a `Wrapper` primitive.

### 7.5.2 Router

Workstations, and increasingly, personal computers, are capable of supporting a rich collection of input devices, many of which can be interchanged (at least conceptually) for particular logical tasks. For example, picking an item on a “2D” display is usually accomplished by mouse, but can also be done using a stylus/tablet combination, or could be done using a more powerful device, e.g. a space-ball, where the feedback is constrained. Switching between multiple streams of input data could be carried out by disconnecting the port of one device from another, and then establishing a new connection. However, connection establishment is a relatively high-cost operation, and is not intended to support this kind of switching. A better way is to establish connections to possible input devices when a network of devices is set up, and then “switch” connections between particular devices. Such a switch, called a `Router` for obvious reasons, is defined in PREMO as a subtype of `MRI_Device`, but also extends the `Controller` object type. Like any `VirtualDevice` derivative, a `Router` has a number of input and output ports. The exact number of such ports is not specified in PREMO and will depend on specific implementations. Internally, the device keeps track of which input ports are connected to which output ports. Data arriving on an input port is copied to all associated output ports. Data arriving at an input port with no associated output port are simply discarded. By extending `Controller`, different combinations of routing between input and output ports can be assigned to specific `Controller` states. Changing a collection of routes can then be done conveniently, by using the state-transition services from `Controller`. The top level of the `Router` interface appears below. Its methods are described separately:

```
public interface Router
  extends MRI_Device, Controller, java.rmi.Remote { }
```

A `Router` must maintain an internal table mapping between ports and (controller) states. Data arriving at an input port may be routed to more than one output port. However, each output port should receive data from at most one input port. No restriction is placed on the amount of fanout that is possible. Subtypes may choose to impose a restriction and broadcast it as a property of such objects. Connections are established by `addConnection`, which defines a connection from an input port to an output port for a given state. It will throw an exception if that output port has already been connected to some input port in that state. An association between input and output ports can be terminated by `dropConnection`. This only requires that the state and output are specified, since the input port is then unique. Both operations will also raise exceptions if either the ports or state specified as parameters are undefined or inappropriate.

```
public void addConnection(String state, int inputPortId,
   int outputPortId)
     throws BadPort, BadState, AlreadyConnected;

public void dropConnection(String state, int outputPortId)
     throws BadPort, BadState, AlreadyConnected;
```

The connections that are defined in a given state can be queried by passing the name of the state to `inquireConnections`. Provided the state is valid, the connections are returned as an array of `Links` objects, each specifying an input port and the output port to which it is connected in that state.

```
public Links[] inquireConnections(String state)
   throws BadState;

public class Links {
   int portA;
   int portB;
};
```

As expected, the definition of the `Links` class is nested within `Router`.

## 7.6 The Scene Database

There are a number of situations where it makes sense to allow multiple devices access to one or more primitives: In a CSCW (Computer Supported Cooperative Work) environment, a group of engineers may be viewing and modifying a shared geometric model for an engineering design. As another example, a shared simulation environment may consist of a relatively static model of some landscape, together with more dynamic descriptions of the avatars for each participant. Furthermore, a single-user application may also benefit from being able to store and access media data via a single interface, rather than for example use a `Modeller` and a `Renderer` to export and import data from an external source such as a local file system.

Explicit shared access to media data is supported by the MRI component through the `Scene` object type, which can be thought of as a database of primitives. `Scene` is a generalisation of the Central Structure Store of PHIGS, or the Scene Database of Open Inventor. The header of the Scene interface is as expected (the methods will be introduced in stages):

```
public interface Scene
  extends VirtualDevice, java.rmi.Remote { }
```

Like other MRI devices, a `Scene` object has a number of ports through which media streams can be received or sent. In the case of `Scene`, these streams are used to place media data into the "database", or to carry such data to a client. The problem that this raises is how clients of a `Scene` object should refer to the data that they wish to access. Manipulation of objects in PREMO is done via object references. Whilst these have been designed from the outset to be transparent regarding distribution, the use of `Scene` as a database conflicts with object persistence. Although the `Scene` type defined by MRI does not specify that objects stored in it may be saved to and retrieved from secondary storage, this is a possibility that subtypes of `Scene` may wish to address. More immedi-

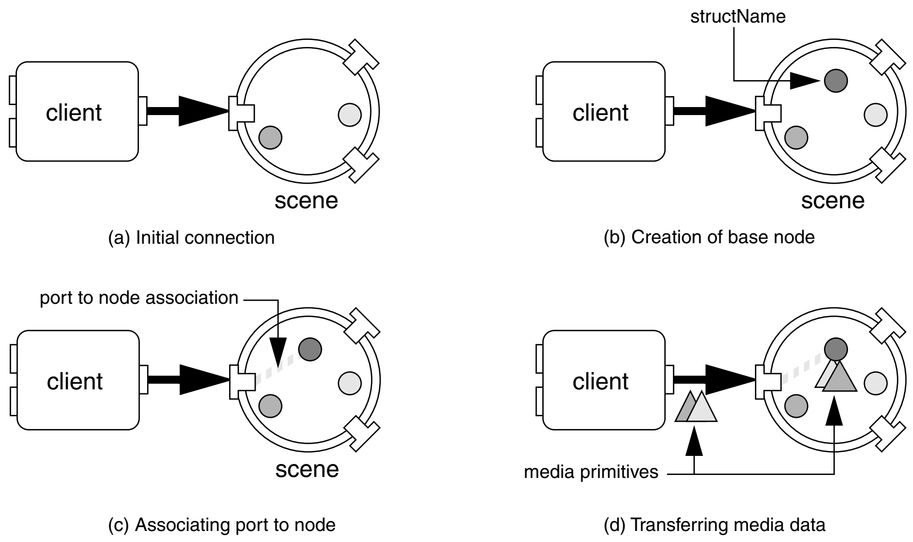


Figure 7-13 — Writing to a `Scene` Object

ately however, a `Scene` could be used as part of a server facility; relying on clients to "discover" references to stored data seems inappropriate. One way of supporting more transparent access would be to have some sort of index or name table, mapping string descriptors to stored primitives. However, rather than construct such a facility from first principles, the design of the `Scene` object makes use of a pre-existing mechanism to name primitives – specifically, the `Name` object associated with each `Structured` primitive.

The MRI `Scene` object thus defines access to media data using `Name` objects as search keys. To retrieve a primitive, a client must request a connection to an output port of a `Scene` object, and also specify a `Name` object that is used to search for a matching `Name` object within a stored primitive. Similarly, media data is stored by defining a `Name` object that will be associated with that data through a `Structured` primitive. It is useful to step through the protocol that a client should use if it wants to store data within a given scene object (the steps are illustrated in Figure 7-13):

1. The client must connect a media stream to an input port, say *P*, of the scene. The mechanisms for doing this are inherited from `MRI_Device`. This is the situation shown in Figure 7-13(a).

2. There are two cases to consider. In the first, the client is going to write data into the scene database and associate that data with a named structured primitive that already exists within the database. At this stage, no further action is needed. If however the structured primitive to which the data should be attached does not yet exist, the client must first create it using the following operation:

```
public void create(Name structName, Object structureType)
        throws AlreadyExists, InvalidType;
```

The first parameter to `create` gives the name to be associated with the structured primitive. The second parameter, `structureType`, is (the name of) the type of object that should contain the name. The value referenced by `structureType` should be the name of a non-abstract object type that is a subtype of `Structured`, for example `Aggregate` or `Sequential`. If `structureType` does not refer to a valid type name, the `InvalidType` exception will be thrown. The `Name` specified by `structName` must be unique; if a matching `Name` object already exists within the database, the `AlreadyExists` exception is thrown.

Figure 7-13(b) shows the effect after a new primitive has been created to act as the base for the stream of data from the client. The other circles within the scene represent primitives that have been created earlier, or by other clients.

3. At this point, we assume that an object exists in the database with a given name (we will see later how this can be checked by a client). The next step for the client is to associate the input port of the `Scene`, to which it will be sending media data, with the object in the scene database where that data will be attached, as shown in Figure 7-13(c). It does this by using the `attachWrite` operation below, giving the `Name` object and the port *P* as parameters.

```
public void attachWrite(Name structName, int portId)
        throws NoStructure, MultiplyDefined, BadPort, AccessFailure;
```

   `structName` must occur exactly once in the database. If it doesn't occur at all, the `NoStructure` exception is thrown; if it occurs more than once, `MultiplyDefined` is thrown. Similarly, the exception `BadPort` is thrown if `portId` does not refer to a valid input port of the device. One further exception can arise. `Scene`, like any multi-user database, has to carry out some form of locking on data to prevent, for example, two clients writing into the same primitive. This issue will be visited again later, but for now we note that the `attachWrite` operation may throw an `AccessFailure` exception if it is not possible for the client to access that primitive for writing due to the activities of other entities in the PREMO application.

4. Once `attachWrite` has completed successfully, any primitive sent by the client along the stream attached to the port *P* will be stored as a component of the structured primitive to which the port is attached. Of course, a primitive transferred in this way may itself be the root of a large collection of media data stored as primitives, and consequently the transfer of data may take a substantial amount of time. The client can discover when all of the data has arrived at the `Scene` by sending a `Tracer` primitive (see Section 7.2.8) after the media data, and asking the event handler of port *P* to notify it when a `Tracer` arrives. Figure 7-13(d) shows this stage of the process.

5. When the client has finished producing media data, it should remove the association between the port from the structured primitive using the `detach` operation:

```
public void detach(int portId) throws BadPort;
```

This will result in a `BadPort` exception if the port referred to by the parameter does not exist. Using `detach` on a port that is not associated with any structured primitive has no effect.

Following `detach`, the connection between the `Scene` port and the client can be removed using the mechanisms provided by the MSS component. If this is done before `detach` has been invoked, the `Scene` object may be left in an undefined state.

Because the protocol for reading data from a `Scene` database is sufficiently similar to that for writing that it should not require illustration, we will just outline the steps involved:

1. A connection is established between the client and an output port of the `Scene` object. As usual, this is done through MSS services.
2. The `attachRead` operation is invoked to associate the output port of the `Scene` object with some structured primitive within the database. The primitive is specified by giving a `Name` object that must match the `Name` object associated with that of a structured primitive. As with `attachWrite`, exceptions are raised if either no occurrence of the `Name` is found, or if more than one occurrence is found.

   ```
   public void attachRead(Name structName, int portId)
      throws NoStructure, MultiplyDefined, BadPort, AccessFailure;
   ```

   Exceptions are also raised if the port specified does not exist (`BadPort`), or if another object is attempting to access the same primitive concurrently (`Access-Failure`).
3. Once the association between port and primitives has been made, the client requests that the primitives attached to the structured node it has referenced be transported from the scene through the specified port. It does this by invoking the `transfer` operation on the `Scene` object. The `portId` parameter identifies which port is being used for the transfer, and thus which primitives are to be transferred.

   ```
   public void transfer(int portId)
      throws BadPort, NotAttached;
   ```

   Following the pattern of previous operations, the `BadPort` exception applies if the designated port is inappropriate. `NotAttached` is thrown if there is no link between the port and a primitive in the database. The `transfer` operation is asynchronous; to indicate completion, the `Scene` will send a single `Tracer` primitive through the output port once the designated primitive has been sent. Arrival of the `Tracer` at the port at the other end of the media stream indicates that the primitive has been transferred. The effect of invoking `transfer` or any other operation that refers to a port in use for reading while the primitive has not been fully read is unspecified.
4. When the reading operation has been completed, the client should invoke `detach` before the MSS services are used to terminate the connection (if that is appropriate at this point).

Two further operations are defined in `Scene` interface. The `delete` operation is given a `Name` object, and will try to remove that object, the structured primitive that contains it, and any component of that structured primitive from the database. Furthermore, it will remove **all** such occurrences of the name from within the database. Note that the name need not belong to a "top level" node in the database, it may be a `Name` object associated with some structured primitive buried within a hierarchy of primitives. Similarly, reading and writing can take place at an arbitrary node within a structured primitive. If no occurrence of the name could be found, the `NoStructure` exception will be thrown. The `Locked` exception will result if the primitive that contains the matching `Name` object is currently attached to some port, either for reading or writing.

```
public void delete(Name structName)
   throws NoStructure, Locked;
```

The remaining operation, `inquireStatus`, can be used by a client to query the state of a given name within the database. It returns one of four values: `NotPresent`, `Locked`, `Available`, and `MultiplyDefined`, which are defined by an enumeration:

```
public final class SceneObjectState extends PREMOEnumeration {
    public static SceneObjectState NotPresent;
    public static SceneObjectState Locked;
    public static SceneObjectState Available;
    public static SceneObjectState MultiplyDefined;
}
```

Note that a name that is multiply defined cannot be locked, since the `attachRead` and `attachWrite` operations require that exactly one `Name` object within the database match that given by the parameter to the operation.

```
   public SceneObjectState inquireStatus(Name structName);
```

Any database-like system that offers concurrent access must address the question of how to prevent multiple readers and writers from interfering with each other. A well known solution to the problem involves systematically locking shared data, allowing only one client to read or write that data at a time. The granularity of locking varies between system and application. A relational database, for example, might provide locking at the level of the database, a relation within the database, or a subset of the rows and columns stored in a particular relation. How an implementation of the `Scene` object type realises locking will depend on the underlying technology. To allow for different strategies, all that the PREMO standard mandates is that a suitable exception will be thrown if an operation cannot complete successfully due to the presence of concurrent access to shared data. Particular locking regimes are free to provide further information about the cause of the locking problem through the fields of the event structure.

## 7.7 Coordination

The `TimeComposite` primitives described in Section 7.2.9.2 provide a simple, declarative model of multimedia content. Any model of multimedia presentation must somehow allow for the aggregation of individual media, so it should not be surprising that, for example, the PREMO `Parallel` primitive plays a similar role to the use of multiple

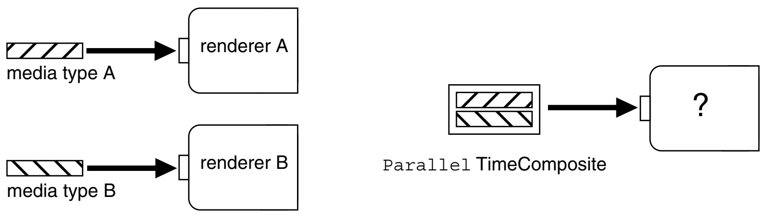


Figure 7-14 — The Problem of New Media Types

tracks in HyTime or channels in MPEG. A HyTime or MPEG player must be able to interpret the particular kind of data carried on each component of the presentation, and render it in a suitable way. For these formats, the problem is simplified because the standard defines the specific kinds of data that a player can be expected to handle. A PREMO system, in contrast, has a more difficult task, since the standard is intended to be extensible. New kinds of media data, and devices for processing such data, can be defined by subtyping. This, however, leaves open the question of how to process a composite presentation containing such data. We may have a device that can process the data in isolation, but not necessarily one that can process this data in combination with other formats. Figure 7-14 illustrates the general problem.

What is needed is a way for a device that processes `TimeComposite` primitives to use devices that are specific to media that might be carried as part of the `TimeComposite`. Such a device could then be configured dynamically to accommodate processors specific to certain media. Upon receiving a `TimeComposite` primitive, it would need to find a device appropriate for each component of the `TimeComposite`, and allocate the components to specific devices. However, that is not all it must do. Different kinds of `TimeComposite` primitive contain implicit and explicit requirements on synchronization between components. If the components are allocated to separate devices, these synchronization constraints must somehow be maintained. This therefore becomes an additional responsibility of the device that manages the distribution of content to processors. Such a device is defined in the PREMO MRI component, and is called a `Coordinator`.

```
public interface Coordinator { }
```

The `Coordinator` addresses three sets of concerns:

1. *Management.* As an MRI device, a `Coordinator` provides a port through which it can receive primitives. It also defines an interface that allows devices (which must be subtypes of `Renderer`) to be added and removed from the `Coordinator`. Here, ‘adding’ a device means registering it as a device that the `Coordinator` can use for processing components of a presentation that it receives as a `TimeComposite` primitive through its input port.

2. *Allocation.* On receiving a `TimeComposite` primitive, a `Coordinator` will assign the primitive, or its components, to one or more of the devices that have been registered with the `Coordinator`.

3. *Synchronization.* A Coordinator is responsible for ensuring that any implicit or explicit synchronization constraints present in any received TimeComposite primitive are respected as the components of the TimeComposite are processed by the devices to which they have been allocated.

Each of these concerns will be examined in more detail in the subsections that follow.

## 7.7.1 Management

Registration and removal of devices to and from a Coordinator is accomplished through two operations in the Coordinator interface. To add a device, the addDevice operation is invoked with a reference to a device, and the identity of the port on that device to which the Coordinator should send primitives. If the device is already registered with the Coordinator, the port specified by the inPortId parameter overrides the port that the Coordinator currently uses. If the port does not correspond to an input port on the device that has an MRI_Format object associated with it, a BadPort exception is raised.

```
public void addDevice(Renderer renderer, int inPortId)
   throws BadPort;
```

Removing a device is simple: the client passes a reference to the device that should be de-registered from a Coordinator as the parameter to dropDevice. There is no effect if the device has not been registered. It is up to the client to ensure that when a device is removed from a Coordinator, it is not ‘in use’ i.e. processing media data. The standard does not define an outcome in the case that the device is ‘in use’.

```
   public void dropDevice(Renderer renderer);
```

One further operation is defined in the Coordinator interface: inquireDevice returns information about the devices registered with the Coordinator as a sequence of pairs, where each pair contains a reference to a registered device and the identity of the port of the device that will be used by the Coordinator. In the Java binding, the sequence is implemented by an array, and the device / port combinations are represented by DeviceInfo objects.

```
public class DeviceInfo {
   public Renderer   renderer;
   public int        inPortId;
}
public DeviceInfo[]  inquireDevice();
```

## 7.7.2 Allocation

Once a Coordinator has received a TimeComposite primitive, it must attempt to allocate it for processing to one or more of the devices that it has at its disposal. In order to accomplish this, it must examine the components of the TimeComposite to determine their constituent media, and then attempt to match this against the information carried in the MRI_Format objects of the device ports that specifies which kinds of primitive that device can accept. The task may be impossible. If a component of the TimeComposite uses a primitive for which no suitable device is available, or if the components of a

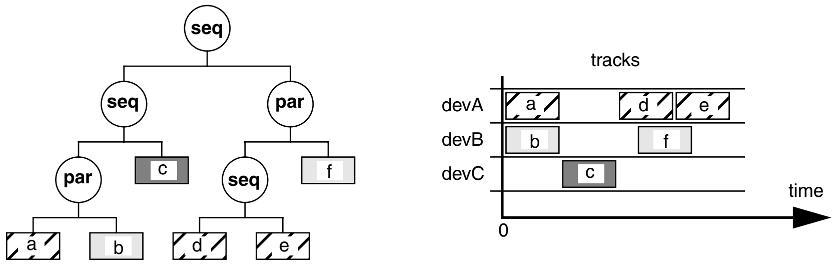


Figure 7-15 — Laying Primitives along Tracks

Parallel primitive require more devices of a particular type than are available, some action must be taken. The Coordinator could abort presentation of the entire TimeComposite, or it could attempt to present those parts for which it has resources available. Different strategies will be appropriate in different applications and circumstances, and for this reason the standard does not mandate what action should be taken if resources are insufficient.

Assuming that the Coordinator's resources are sufficient, the task of allocation can be viewed as laying out media primitives along tracks, one per device that the Coordinator has available to it. As shown in Figure 7-15, the arrangement of primitives along tracks should reflect the organisation of primitives within the Sequential, Parallel and Alternate TimeComposites that describe the presentation. Note that the choice of component to be processed from an Alternate primitive must be made effectively at the point that this allocation is carried out. Although not specifically described in the standard, it is in principle possible that a Coordinator could be allowed, by changing the state of an Alternate primitive, to select a component that will make the allocation feasible or particularly efficient.

### 7.7.3 Synchronization

Once the components of a TimeComposite have been scheduled for processing, the Coordinator should place the primitives into media streams that are linked to the designated input ports of the rendering devices. At this point, the processing of the media data becomes a concurrent activity, employing a collection of independent devices. The Coordinator is effectively acting as the client of its 'local' resources, and as such has to take responsibility for ensuring that the processes realised by the devices are appropriately synchronized. The synchronization constraints that the Coordinator must fulfill include (see Section 7.2.9 for details of the primitives mentioned):

1. Respecting the startSync and endSync flags of Parallel TimeComposite primitives
2. Taking into account the compStart and compEnd offsets of all TimeComposite primitives, and the startDelta and endDelta offsets of Sequential TimeComposites
3. Respecting the min / max duration for the overall TimeComposite.

The PREMO standard does not mandate any specific strategy for meeting these requirements but it does make three suggestions as to how synchronization might be realised, which are described here:

1. The input port of each device used by the Coordinator has an associated StreamControl object. The finite-state-machine (Controller object) embedded within the StreamControl object can be used by the Coordinator to start, stop, pause and resume the processing activity of each device. The Coordinator can acquire a certain level of feedback about progress through the inquiry operations provided through StreamControl. However, at the level of stopping and starting processing for entire primitives, a simpler approach for the Coordinator is to insert a Tracer primitive into each stream immediately following the media primitive, and arrange for it to be notified when the Tracer is processed at the input port of each device.
2. The StreamControl objects at the device input ports can also be used to give a finer level of inter-media synchronization, by using SynchronizationElements along the progression space, coupled with an ANDSynchronizationPoint object, as described in Section 5.5 on page 90. Suitable points for applying this strategy are the time points corresponding to the start and end of segments of primitive data. When used in conjunction with the Tracer primitives as described above, the synchronization elements may be offset to account for the presence of the Tracer, although as mentioned in Section 7.2.8, the time taken to process a Tracer will generally be small compared to the time taken to process a media primitive.
3. Fine control over synchronization between multiple media streams can be achieved by the Coordinator by placing periodic synchronization elements along the progression space of the StreamControl objects at the device ports.

The options are clearly not mutually exclusive, nor are they in any sense ‘complete’. Figure 7-16 illustrates how the first and second of these approaches might be combined although fine-grained synchronization through periodic synchronization elements was omitted for clarity. The “tracks” are intended to represent the progression space associated with the input port of each device. Note that the relative size of Tracer primitives to other media primitives is much larger in the figure than would be expected in practice.

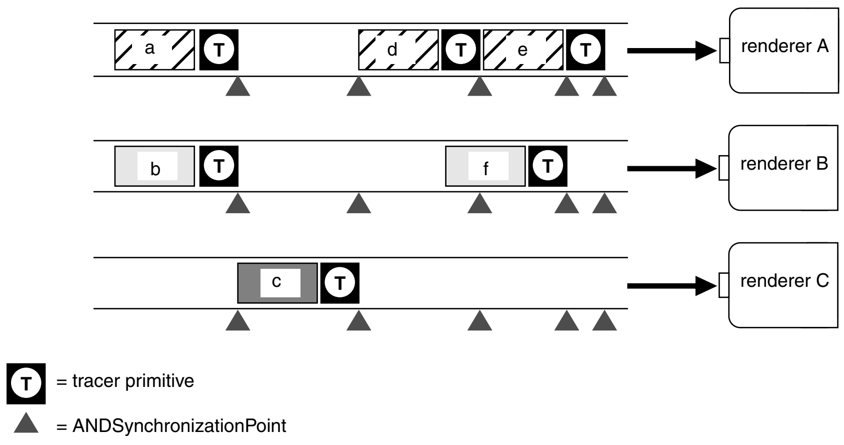


Figure 7-16 — Synchronization of Coordinated Streams

# Chapter 8

# Detailed Java Specifications of the PREMO Objects

## 8.1 Introduction

This chapter collects all PREMO object specifications. No detailed explanation for the interfaces, classes, or methods are provided here, the reader should refer to the main text.

In the case of classes, only the public part of the interface is listed. In the case of interfaces, some methods are repeated in interface extensions although, formally, this would not be necessary; repeated methods refer to the fact that the interface implementation represents a major change or extension in semantics for that specific method.

The package and import statements are not listed to avoid cluttering the text.

## 8.2 Foundation Objects

All the classes and interfaces are in the package `premo.std.part2`. All classes import the package `premo.impl.utils`, to refer to, e.g., the `PREMOEnumeration` class.

### 8.2.1 Enumerations

Only the relevant constants for enumerations are listed. The special construction mechanism, common to all PREMO enumerations, is omitted; see page 62 for further details.

```
public final class ActionType extends PREMOEnumeration {
   public static ActionType Enter;
   public static ActionType Leave;
}

public final class AndOr extends PREMOEnumeration {
   public static AndOr And;
   public static AndOr Or;
}

public final class Direction extends PREMOEnumeration {
   public static Direction Forward;
   public static Direction Backward;
}
```

```
public final class TimeUnit extends PREMOEnumeration {
    public static TimeUnit Picoseconds;
    public static TimeUnit Nanoseconds;
    public static TimeUnit Microseconds;
    public static TimeUnit Miliseconds;
    public static TimeUnit Second;
    public static TimeUnit Minute;
    public static TimeUnit Hour;
    public static TimeUnit Day;
    public static TimeUnit Month;
    public static TimeUnit Year;
}

public final class ConstraintOp extends PREMOEnumeration {
   public static ConstraintOp Equal;
   public static ConstraintOp NotEqual;
   public static ConstraintOp GreaterThan;
   public static ConstraintOp GreaterThanOrEqual;
   public static ConstraintOp LessThan;
   public static ConstraintOp LessThanOrEqual;
   public static ConstraintOp Prefix;
   public static ConstraintOp Suffix;
   public static ConstraintOp NotPrefix;
   public static ConstraintOp NotSuffix;
   public static ConstraintOp Includes;
   public static ConstraintOp Excludes;
}
```

### 8.2.2 Additional Data Types

```
// Integer codes for Synchronizable and Timer states
// Usage of integer codes make the merge and extension of
// states easier.
public final class State implements java.io.Serializable
{
    public static final int TSTOPPED = 0;
    public static final int TSTARTED = 1;
    public static final int TPAUSED  = 2;
    public static final int STOPPED  = 0;
    public static final int STARTED  = 1;
    public static final int PAUSED   = 2;
    public static final int WAITING  = 3;
}
```

### 8.2.3 Top Level of PREMO Hierarchy

```
// See section 5.3.1 on page 64
public interface PREMOObject extends java.rmi.Remote {
   java.lang.Class inquireType()
      throws java.rmi.RemoteException;
   java.lang.Class[] inquireTypeGraph()
      throws java.rmi.RemoteException;
   java.lang.Class[] inquireImmediateSupertypes()
      throws java.rmi.RemoteException;
}

// See section 5.3.2 on page 65; also, section A.1.2 on page 232
public abstract class SimplePREMOObject
implements /* PREMOObject, */ java.io.Serializable {
   java.lang.Class inquireType()
      throws java.rmi.RemoteException;
   java.lang.Class[] inquireTypeGraph()
      throws java.rmi.RemoteException;
   java.lang.Class[] inquireImmediateSupertypes()
      throws java.rmi.RemoteException;
}

// See section 5.3.3 on page 68
public interface Callback extends PREMOObject {
   void callback(Event callbackValue)
      throws java.rmi.RemoteException;
}

// See section 5.3.3 on page 68
public interface CallbackByName extends PREMOObject {
   void callback(Event callbackValue)
      throws OperationNotDefined, java.rmi.RemoteException;
}

// See section 5.3.4 on page 69
public interface EnhancedPREMOObject
extends PREMOObject, java.rmi.Remote {
   void defineProperty(String key, Object[] value)
      throws ReadOnlyProperty, java.rmi.RemoteException;
   void undefineProperty(String key)
      throws ReadOnlyProperty, NoKey, java.rmi.RemoteException;
   void addValue(String key, Object value)
      throws ReadOnlyProperty, java.rmi.RemoteException;
   void removeValue(String key, Object value)
      throws   ReadOnlyProperty, NoKey, InvalidValue,
               java.rmi.RemoteException;

   public static class KeyInfo implements java.io.Serializable {
         public String  key;
         public boolean readOnly;
   }
```

```
    KeyInfo[] inquireProperties()
        throws java.rmi.RemoteException;
    Object[] getProperty(String key)
        throws NoKey, java.rmi.RemoteException;
    PropertyPair[] getPairs()
        throws java.rmi.RemoteException;

    public static class MatchPropertyResults
        implements java.io.Serializable {
            public PropertyPair[] satisfied;
            public PropertyPair[] unsatisfied;
    }
    MatchPropertyResults matchProperties(Constraint[] constraintList)
        throws java.rmi.RemoteException;

    void setPropertyCallback( String key, Callback callback,
                              String eventName )
        throws NoKey, java.rmi.RemoteException;
}
```

### 8.2.4 Structures

```
// This structure does not appear in the PREMO document directly,
// but is used by several objects; it has been abstracted out
// as a separate class
public final class PropertyPair implements java.io.Serializable {
    public String   key;
    public Object[] value;
    public PropertyPair( String k, Object[] v);
    public PropertyPair();
}

// See page 83
public class ActionElement
extends SimplePREMOObject implements java.io.Serializable {
    public Callback eventHandler;
    public String eventName;
}

// See section 5.3.2.2 on page 67
public class Constraint
extends SimplePREMOObject implements java.io.Serializable {
    public ConstraintOp constraintOp = ConstraintOp.Equal;
    public static class KeyValue implements java.io.Serializable {
        public String key = "";
        public Object value;
    }
    public KeyValue    keyValue = new KeyValue();
}
```

```
// See section 5.3.2.1 on page 66
// "equals" overrides the corresponding methods in
// java.lang.Object; "clone" implements the Cloneable interface

public class Event extends SimplePREMOObject
implements java.lang.Cloneable, java.io.Serializable{

   public static class EventData implements java.io.Serializable {
      public String           key   = "";
      public java.lang.Object value;
      public boolean equals(java.lang.Object otherO);
   }

   public String              eventName = "";
   public EventData[]         eventData = new EventData[0];
   public EnhancedPREMOObject eventSource;
   public boolean equals(java.lang.Object otherE);
   public java.lang.Object clone();
}

// See section 5.5.1.3.3 on page 101

public class SyncElement
extends SimplePREMOObject implements java.io.Serializable {

   public Callback eventHandler;
   public Event    syncEvent;
   public boolean  waitFlag;
}
```

### 8.2.5 General Utility Objects

#### 8.2.5.1 Event Management

```
// See section 5.4.1.2 on page 76

public interface EventHandler
extends Callback, EnhancedPREMOObject, java.rmi.Remote {

   public long register(   String eventType, Constraint[] constrains,
                           AndOr matchMode, Callback theCallback)
      throws java.rmi.RemoteException;
   public void unregister(long id)
      throws InvalidEventId, java.rmi.RemoteException;
   public void dispatchEvent(Event e)
      throws java.rmi.RemoteException;
}
```

```
// See section 5.4.1.3 on page 78
public interface SynchronizationPoint
extends EventHandler, java.rmi.Remote {
   public void addSyncEvent(Event e)
      throws RepeatedEvent, java.rmi.RemoteException;
   public void deleteSyncEvent(Event e)
      throws UnknownEvent, java.rmi.RemoteException;

   public long register(   String eventType, Constraint[] constrains,
                           AndOr matchMode, Callback theCallback)
      throws InvalidEventId, java.rmi.RemoteException;
   public void dispatchEvent(Event e)
      throws UnknownEvent, java.rmi.RemoteException;
}


// See section 5.4.1.3 on page 78
public interface ANDSynchronizationPoint
extends SynchronizationPoint, java.rmi.Remote {
   public void addSyncEvent(Event e)
      throws RepeatedEvent, java.rmi.RemoteException;
   public void deleteSyncEvent(Event e)
      throws UnknownEvent, java.rmi.RemoteException;

   public void dispatchEvent(Event e)
      throws UnknownEvent, java.rmi.RemoteException;
}
```

#### 8.2.5.2 Controllers

```
// See section 5.4.2 on page 81
public interface Controller
extends Callback, EnhancedPREMOObject, java.rmi.Remote {
   public String getCurrentState()
      throws java.rmi.RemoteException;
   public String[] getPossibleStates()
      throws java.rmi.RemoteException;

   public void handleEvent(Event e)
      throws java.rmi.RemoteException;

   public void setAction(  String state, ActionElement action,
                           ActionType aType)
      throws WrongState, java.rmi.RemoteException;
   public void removeAction(String state, ActionType aType)
      throws WrongState, java.rmi.RemoteException;

   public void setActionOnPair(  String stateOld, String stateNew,
                                 ActionElement action)
        throws WrongState, java.rmi.RemoteException;
   public void removeActionOnPair(String stateOld, String stateNew)
        throws WrongState, java.rmi.RemoteException;
}
```

#### 8.2.5.3 Time Objects

```
// See section 5.4.3.2 on page 88
public interface Clock
extends EnhancedPREMOObject, java.rmi.Remote {
   TimeUnit getTickUnit()           throws java.rmi.RemoteException;
   void setTickUnit(TimeUnit unit)  throws java.rmi.RemoteException;
   TimeUnit getAccuracyUnit()       throws java.rmi.RemoteException;
   void setAccuracyUnit(TimeUnit unit)
      throws java.rmi.RemoteException;

   long getAccuracy()               throws java.rmi.RemoteException;
   long inquireTick()               throws java.rmi.RemoteException;
}


public interface SysClock extends Clock, java.rmi.Remote {
   long inquireTick()               throws java.rmi.RemoteException;
}


public interface Timer extends Clock, java.rmi.Remote {
   int getTimerCurrentState() throws java.rmi.RemoteException;

   void start()               throws java.rmi.RemoteException;
   void stop()                throws java.rmi.RemoteException;
   void pause()               throws java.rmi.RemoteException;
   void resume()              throws java.rmi.RemoteException;
   void reset();              throws java.rmi.RemoteException;

   long inquireTick()         throws java.rmi.RemoteException;
}
```

### 8.2.6 Sychronization Objects

```
// See section 5.5.1.3 on page 99
public interface Synchronizable
extends CallbackByName, EnhancedPREMOObject, java.rmi.Remote {
   int getCurrentState()          throws java.rmi.RemoteException;
   Number getCurrentPosition()    throws java.rmi.RemoteException;
   Number getMinimumPosition()    throws java.rmi.RemoteException;
   Number getMaximumPosition()    throws java.rmi.RemoteException;
   int getLoopCounter()           throws java.rmi.RemoteException;

   void setDirection(Direction where)
      throws WrongState, java.rmi.RemoteException;
   Direction getDirection()
      throws java.rmi.RemoteException;

   void setStartPosition(Number position)
      throws   WrongValue, WrongState, IllegalArgumentException,
               java.rmi.RemoteException;
   Number getStartPosition()
      throws java.rmi.RemoteException;
```

```
void setEndPosition(Number position)
   throws   WrongValue, WrongState, IllegalArgumentException,
            java.rmi.RemoteException;
Number getEndPosition() throws java.rmi.RemoteException;

void setRepeatFlag(boolean flag)
   throws WrongState, java.rmi.RemoteException;
boolean getRepeatFlag() throws java.rmi.RemoteException;

void setNLoop(int value)
   throws   WrongState, IllegalArgumentException,
            java.rmi.RemoteException;
int getNLoop() throws java.rmi.RemoteException;

void resetLoopCounter()
   throws WrongState, java.rmi.RemoteException;

void jump(Number position)
   throws   WrongState, WrongValue, IllegalArgumentException,
            java.rmi.RemoteException;

void start()   throws WrongState, java.rmi.RemoteException;
void stop()    throws java.rmi.RemoteException;
void pause()   throws WrongState, java.rmi.RemoteException;
void resume()  throws WrongState, java.rmi.RemoteException;

void setSyncElement(Number position, SyncElement syncElement)
   throws   WrongState, WrongValue, IllegalArgumentException,
            java.rmi.RemoteException;
void deleteSyncElement(Number position)
   throws   WrongState, WrongValue, IllegalArgumentException,
            java.rmi.RemoteException;

public class SyncInfo {
   public SyncElement syncElement;
   public Number      position;
}
SyncInfo[] getSyncElements(Number posMin, Number posMax)
   throws   WrongValue, IllegalArgumentException,
            java.rmi.RemoteException;

void setPeriodicSyncElement(  Number startRefPoint,
                              Number endRefPoint,
                              Number periodicity,
                              SyncElement syncData)
   throws   WrongState, WrongValue, IllegalArgumentException,
            java.rmi.RemoteException;
void deletePeriodicSyncElement(  Number startRefPoint,
                                 Number endRefPoint,
                                 Number periodicity)
   throws   WrongState, WrongValue, IllegalArgumentException,
            java.rmi.RemoteException;
   void setActionOnPair(   int stateOld, int stateNew,
                           ActionElement action)
      throws  WrongState, java.rmi.RemoteException;
   void removeActionOnPair(int stateOld, int stateNew)
      throws  WrongState, java.rmi.RemoteException;

   void clearSyncElements()
      throws WrongState, java.rmi.RemoteException;
}


// See section 5.5.2 on page 103
public interface TimeSynchronizable
extends Synchronizable, Timer, java.rmi.Remote {
   void setSpeed(double speed)
      throws WrongState, java.rmi.RemoteException;
   double getSpeed()
      throws java.rmi.RemoteException;

   long getTimeCurrentPosition() throws java.rmi.RemoteException;
   long getTimeMinimumPosition() throws java.rmi.RemoteException;
   long getTimeMaximumPosition() throws java.rmi.RemoteException;
   void setTimeStartPosition(long position)
      throws   WrongValue, WrongState, IllegalArgumentException,
               java.rmi.RemoteException;
   long getTimeStartPosition()
      throws java.rmi.RemoteException;
   void setTimeEndPosition(long position)
      throws   WrongValue, WrongState, IllegalArgumentException,
               java.rmi.RemoteException;
   long getTimeEndPosition()     throws java.rmi.RemoteException;
   void jump(long position)
      throws  WrongState, WrongValue, java.rmi.RemoteException;
   void setSyncElement(long position, SyncElement syncElement)
      throws  WrongState, WrongValue, java.rmi.RemoteException;
   void deleteSyncElement(long position)
      throws  WrongState, WrongValue, java.rmi.RemoteException;
   public class TimeSyncInfo {
      public SyncElement syncElement;
      public long        position;
   }
   TimeSyncInfo[] getSyncElements(long posMin, long posMax)
      throws  WrongValue, java.rmi.RemoteException;
   void setPeriodicSyncElement(  long startRefPoint, long endRefPoint,
                                 long periodicity,
                                 SyncElement syncData)
      throws  WrongState, WrongValue, java.rmi.RemoteException;
   void deletePeriodicSyncElement(  long startRefPoint,
                                    long endRefPoint,
                                    long periodicity)
      throws  WrongState, WrongValue, java.rmi.RemoteException;
   void play()    throws WrongState, java.rmi.RemoteException;
   void run()     throws WrongState, java.rmi.RemoteException;
   void stop()    throws java.rmi.RemoteException;
   void pause()   throws WrongState, java.rmi.RemoteException;
   void resume()  throws WrongState, java.rmi.RemoteException;

   Number timeToSpace(long time)
      throws java.rmi.RemoteException;
   long spaceToTime(Number position)
      throws IllegalArgumentException, java.rmi.RemoteException;
}

// See section 5.5.3 on page 107
public interface TimeSlave
extends  TimeSynchronizable, java.rmi.Remote {
   void setMaster(TimeSynchronizable master)
      throws WrongState, java.rmi.RemoteException;
   TimeSynchronizable getMaster()
      throws java.rmi.RemoteException;
   long inquireAlignment()    throws java.rmi.RemoteException;

   public class syncHandler {
      Callback    handler;
      long        threshold;
   }
   void setSyncEventHandlers( syncHandler[] syncEventHandlers )
      throws java.rmi.RemoteException;
}

// See section 5.5.4 on page 109
public interface TimeLine
extends  TimeSynchronizable, java.rmi.Remote {
   void setSpeed(double speed)
      throws WrongState, java.rmi.RemoteException;
}
```

### 8.2.7 Negotiation and Configuration Management

```
   // See section 5.6.2 on page 113
   public interface PropertyInquiry
   extends EnhancedPREMOObject, java.rmi.Remote {
      java.lang.Object[] inquireNativePropertyValue(String key)
         throws java.rmi.RemoteException, InvalidKey;
   }
```

```
// See section 5.6.3 on page 114
public interface PropertyConstraint
extends PropertyInquiry, java.rmi.Remote {
   void defineProperty(String key, Object[] value)
      throws   ReadOnlyProperty, InvalidValue,
               java.rmi.RemoteException;
   public void addValue(String key, Object value)
      throws   ReadOnlyProperty, InvalidValue,
               java.rmi.RemoteException;
   public void bind()
      throws InvalidValue, java.rmi.RemoteException;
   public void unbind()
      throws java.rmi.RemoteException;
   public PropertyPair[] constrain( PropertyPair[] constraints )
      throws InvalidKey, InvalidValue, java.rmi.RemoteException;
   public PropertyPair[] select( PropertyPair[] constraints )
      throws InvalidKey, InvalidValue, java.rmi.RemoteException;
}
```

### 8.2.8 Creation of Service Objects

```
// See section 5.7.1 on page 118
public interface GenericFactory
extends PropertyInquiry, java.rmi.Remote
{
   PropertyInquiry createObject( String            objectType,
                                 PropertyPair[]    constraints,
                                 Object            initValue)
      throws   InvalidCapabilities, CannotMeetCapabilities,
               InvalidType, IncorrectInit, java.rmi.RemoteException;
   PropertyInquiry createObject( java.lang.Class   objectType,
                                 PropertyPair[]    constraints,
                                 Object            initValue)
      throws   InvalidCapabilities, CannotMeetCapabilities,
               InvalidType, IncorrectInit, java.rmi.RemoteException;
}

// See section 5.7.2 on page 120
public interface FactoryFinder
extends EnhancedPREMOObject, java.rmi.Remote {
   GenericFactory[]
      findFactories( String            objectType,
                     PropertyPair[]    objectConstrains,
                     PropertyPair[]    factoryConstrains )
      throws   InvalidCapabilities, CannotMeetCapabilities,
               InvalidType, java.rmi.RemoteException;
   GenericFactory[]
      findFactories( java.lang.Class   objectType,
                     PropertyPair[]    objectConstrains,
                     PropertyPair[]    factoryConstrains )
      throws   InvalidCapabilities, CannotMeetCapabilities,
               InvalidType, java.rmi.RemoteException;
}
```

## 8.3 Multimedia Systems Services

All the classes and interfaces are in the package `premo.std.part3`. All classes import the packages `premo.impl.utils` (to refer to, e.g., the `PREMOEnumeration` class), and `premo.std.part2`.

### 8.3.1 Enumerations

Only the relevant constants for enumerations are listed. The special construction mechanism, common to all PREMO enumerations, is omitted; see page 62 for further details.

```
// See section 6.5.1.2 on page 147

public final class PortType
extends PREMOEnumeration implements java.io.Serializable
{
    public static PortType INPUT;
    public static PortType OUTPUT;
}
```

### 8.3.2 Structures and Additional Data Types

```
// Integer codes for Synchronizable and Timer states
// Usage of integer codes make the merge and extension of
// states easier.

public final class MSS_State implements java.io.Serializable
{
   public static final int    MUTED    = 4;
   public static final int    PRIMING  = 5;
   public static final int    DRAINING = 6;
}


// See section 6.4.1 on page 141

public final class ConfInfo implements java.io.Serializable
{
   public String  semName;
   public Class   objectType;
   public ConfInfo(String name, Class type);
   public ConfInfo();
}
```

```
// See section 6.5.1.2 on page 147

public class PortConfig
extends SimplePREMOObject implements java.io.Serializable
{
   public ConfInfo      qos;
   public Callback      eventHandler;
   public StreamControl streamControl;
   public ConfInfo      protocol;
   public static class formatData implements java.io.Serializable {
      public long       time;
      public ConfInfo   name;
   }
   public formatData[]  formats;
}
```

## 8.3.3 Configuration Objects

```
// See section 6.2.1 on page 131
public interface Format
extends PropertyConstraint, java.rmi.Remote {}


// See section 6.2.2 on page 132
public interface MultimediaStreamProtocol
extends PropertyConstraint, java.rmi.Remote {}


// See section 6.2.2 on page 132
public interface InterNodeTransport
extends MultimediaStreamProtocol, java.rmi.Remote {}


// See section 6.2.2 on page 132
public interface InterNodeTransport
extends MultimediaStreamProtocol, java.rmi.Remote {}


// See section 6.2.2 on page 132
public interface IntraNodeTransport
extends MultimediaStreamProtocol, java.rmi.Remote {}


// section 6.2.3 on page 134
public interface QoSDescriptor
extends PropertyConstraint, java.rmi.Remote {}
```

### 8.3.4 Stream Control

```
// See section 6.3.1 on page 136
public interface StreamControl
extends TimeSynchronizable, java.rmi.Remote
{
   int getCurrentState() throws java.rmi.RemoteException;
   void mute()  throws WrongState, java.rmi.RemoteException;
   void prime() throws WrongState, java.rmi.RemoteException;
   void drain() throws WrongState, java.rmi.RemoteException;
}


// See section 6.3.2 on page 140
public interface SyncStreamControl
extends TimeSlave, SyncControl java.rmi.Remote {}
```

### 8.3.5 Virtual Resource

```
// See section 6.4 on page 140
public interface VirtualResource
extends PropertyInquiry, java.rmi.Remote {
   void setResourceEventHandler(Callback e)
      throws java.rmi.RemoteException;
   Callback getResourceEventHandler()
      throws java.rmi.RemoteException;

   StreamControl getStreamControl() throws java.rmi.RemoteException;

   PropertyConstraint resolve(String semName)
      throws InvalidName, java.rmi.RemoteException;

   void acquireResource()
      throws ResourceNotAvailable, java.rmi.RemoteException;
   void releaseResource() throws java.rmi.RemoteException;

   public class ProposedValues implements java.io.Serializable {
      public String          semanticName;
      public PropertyPair[]  replacement;
   }
   public class ValidationResult implements java.io.Serializable {
      public boolean         result;
      public ProposedValues  proposedValues;
   }
   ValidationResult validate() throws java.rmi.RemoteException;
}
```

### 8.3.6 Virtual Device

```
// See section 6.5 on page 146
public interface VirtualDevice
extends VirtualResource, java.rmi.Remote {
   void acquireResource()
      throws ResourceNotAvailable, java.rmi.RemoteException;
   void releaseResource() throws java.rmi.RemoteException;

   int[] getPorts() throws java.rmi.RemoteException;

   public static class PortDescr implements java.io.Serializable {
      public PortConfig config;
      public PortType   type;
   }
   PortDescr getPortConfig(int portId)
      throws InvalidPort, java.rmi.RemoteException;
   void setPortConfig(int portId, PortConfig portConfig)
      throws InvalidPort, java.rmi.RemoteException;

   ValidationResult portValidate(PortType port, String formatName)
      throws InvalidName, InvalidPort;

   VirtualConnection getConnection(int PortId)
      throws InvalidPort, java.rmi.RemoveException;
}
```

### 8.3.7 Virtual Connections

```
// See section 6.6 on page 155
public interface VirtualConnection
extends VirtualResource, java.rmi.Remote {
   void connect(  VirtualDevice master, int portMaster,
                  VirtualDevice slave,  int portSlave)
      throws   ConfigurationMismatch, PortMismatch,
               ResourceNotAvailable, InvalidPort,
               java.rmi.RemoteException;

   void disconnect() throws java.rmi.RemoteException;

   public class EndpointInfo {
      public VirtualDevice device;
      public int           port;
      public boolean       isMaster;
   }
   EndpointInfo[] getEndpointInfoList()
      throws java.rmi.RemoteException;
}
```

```
// See section 6.6.4 on page 160
public interface VirtualConnectionMulticast
extends VirtualConnection, java.rmi.Remote {
   void attach(VirtualDevice device,  int portID)
      throws   ConfigurationMismatch, PortMismatch,
               ResourceNotAvailable, InvalidPort,
               java.rmi.RemoteException;

   void detach(VirtualDevice device,  int portID)
      throws PortMismatch, java.rmi.RemoteException;
}
```

### 8.3.8 Group

```
// See section 6.7 on page 161
public interface Group
extends VirtualResource, java.rmi.Remote {
   void acquireResource()
      throws ResourceNotAvailable, java.rmi.RemoteException;
   void releaseResource() throws java.rmi.RemoteException;

   void addResource(VirtualResource resource)
      throws java.rmi.RemoteException;
   void removeResource(VirtualResource resource)
      throws ResourceNotAvailable, java.rmi.RemoteException;

   void addResourceGraph(VirtualResource resource)
      throws java.rmi.RemoteException;
   void removeResourceGraph(VirtualResource resource)
      throws ResourceNotAvailable, java.rmi.RemoteException;

   VirtualResource[] getResourceList()
      throws java.rmi.RemoteException;

}
```

### 8.3.9 Logical Device

```
// See section 6.8 on page 163
public interface LogicalDevice
extends VirtualDevice, Group, java.rmi.Remote {
   int definePort(VirtualDevice refVirtualDevice, int portId)
      throws InvalidPort, InvalidDevice, java.rmi.RemoteException;
}
```

## 8.4 The Modelling, Rendering, and Interaction Component

All the classes and interfaces are in the package `premo.std.part4.`

### 8.4.1 Objects for Coordinate Spaces

#### 8.4.1.1 Coordinate Object

```
// See section 7.3.1 on page 186
public abstract class Coordinate extends SimplePREMOObject
{
   static int dimensionality;
   abstract public int[] getRange(int dimension);
   abstract public void setComponent(int dimension, int value);
   abstract public int getComponent(int dimension);
}
```

#### 8.4.1.2 Colour Object

```
// See section 7.3.3 on page 187
abstract public class Colour extends Coordinate
{
   String colourModel;
}
```

#### 8.4.1.3 TimeLocation Object

```
// See section 7.3.2 on page 187
abstract public class TimeLocation extends Coordinate{
    static {
            dimensionality = 1;
    }
}
```

### 8.4.2 Name Object

```
// See section 7.2.10 on page 185
public abstract class Name extends SimplePREMOObject
{
    String[] tag;
    public boolean equal(Name otherName);
}
```

### 8.4.3 Objects for Media Primitives

#### 8.4.3.1 Primitive Object

```
// See section 7.2 on page 167
public class Primitive extends SimplePREMOObject{}
```

#### 8.4.3.2 Captured Object

```
// See section 7.2.3 on page 170
public class Captured extends Primitive
{
    protected VirtualDevice srcDevice;
    protected int srcPort;
}
```

#### 8.4.3.3 Primitives with Spatial and/or Temporal Form

##### *8.4.3.3.1 Form object*

```
// See section 7.2.4 on page 171
public class Form extends Primitive{}
```

#### 8.4.3.4 Form Primitives for Audio Media Data

##### *8.4.3.4.1 Audio Object*

```
// See page 171
public class Audio extends Form
{
    int instrument;
    int[] score;
}
```

##### *8.4.3.4.2 Music Object*

```
// See page 171
public class Music extends Audio
{
    public int instrument;
    public int score;
}
```

#### *8.4.3.4.3 Speech Object*

```
// See page 171
public class Speech extends Audio
{
    public VocalCharacteristics voice;
    public char[] text;
}
```

### 8.4.3.5 Form Primitives for Geometric Media Data

#### *8.4.3.5.1 Geometric Object*

```
// See section 7.2.4 on page 171
public class Form extends Primitive{}
```

#### *8.4.3.5.2 Tactile Object*

```
// See section 7.2.5 on page 172
abstract public class Tactile extends Form{}
```

#### *8.4.3.5.3 Text Object*

```
// See page 173
public class Text extends Form{}
```

### 8.4.3.6 Primitives for the Modification of Media Data

#### *8.4.3.6.1 Modifier*

```
// See section 7.2.6 on page 173
abstract public class Modifier extends Primitive{}
```

### 8.4.3.7 Modifier Primitives for Audio Media Data

#### *8.4.3.7.1 Acoustic Object*

```
// See page 171
abstract public class Acoustic extends Modifier{}
```

#### *8.4.3.7.2 SoundCharacteristic Object*

```
// See page 171
abstract public class SoundCharacteristic
    extends Acoustic{};
```

#### *8.4.3.7.3 VocalCharacteristic Object*

```
// See page 171
abstract public class VocalCharacteristic
    extends Acoustic{};
```

### 8.4.3.8 Modifier Primitives for Structural Aspects of Media Data

#### *8.4.3.8.1 Structural Object*

```
// See page 174
abstract public class Structural
    extends Modifier{};
```

#### *8.4.3.8.2 Transformation Object*

```
// See page 174
abstract public class Transformation extends Structural{}
```

#### *8.4.3.8.3 Constraint Object*

```
// See page 174
abstract public class Constraint extends Structural{}
```

#### *8.4.3.8.4 TimeFrame Object*

```
// See page 175
public class TimeFrame extends Modifier{}
```

### 8.4.3.9 Modifier Primitives for Visual Aspects of Media Data

#### *8.4.3.9.1 Visual Object*

```
// See page 175
abstract public class Visual extends Modifier{}
```

#### *8.4.3.9.2 Light Object*

```
// See page 175
abstract public class Light extends Visual{}
```

#### *8.4.3.9.3 Material Object*

```
// See page 175
abstract public class Material extends Visual{}
```

#### *8.4.3.9.4 Shading Object*

```
// See page 175
abstract public class Shading extends Visual{}
```

#### *8.4.3.9.5 Texture Object*

```
// See page 175
abstract public class Texture extends Visual{}
```

#### *8.4.3.9.6 Reference Object*

```
// See section 7.2.10 on page 185
public class Reference extends Primitive{}
```

### 8.4.3.10 Organising Primitives into Structures

#### *8.4.3.10.1 Structured Object*

```
// See section 7.2.9 on page 177
public abstract class Structured extends Primitive
{
    Primitive[] components;
    Name label;
}
```

#### *8.4.3.10.2 Aggregate Object*

```
// See section 7.2.9.1 on page 177
public abstract class Aggregate extends Structured{}
```

### 8.4.3.11 Organising Media Data within Time

#### *8.4.3.11.1 TimeComposite Object*

```
// See section 7.2.9.2 on page 179
abstract public class TimeComposite extends Structured
{
   long min, max;
   long startTime, endTime;
   Callback monitor;
}
```

#### *8.4.3.11.2 Sequential Object*

```
// See section 7.2.9.2.1 on page 181
public class Sequential extends TimeComposite
{
   long startDelta, endDelta;
   OverlapType overlap;
}
```

#### *8.4.3.11.3 Parallel Object*

```
// See section 7.2.9.2.2 on page 182
public class Parallel extends TimeComposite
{
   boolean startSync, endSync;
}
```

#### *8.4.3.11.4 Alternate Object*

```
// See section 7.2.9.2.3 on page 184
public class Alternate extends TimeComposite
{
   Controller selector;
   AlternateSequence[] options;
}
```

#### *8.4.3.11.5 Tracer Object*

```
// See section 7.2.8 on page 176
public class Tracer extends Primitive
{
   public Event trace;
   public Tracer(Event trace)
   {
      this.trace = trace;
      trace.eventSource = (premo.std.part2.EnhancedPREMOObject)this;
   }
}
```

#### *8.4.3.11.6 Wrapper Object*

```
// See section 7.2.7 on page 176
public class Wrapper extends Primitive
{
   Object content;
}
```

### 8.4.4 Objects for Describing Properties of Devices

#### 8.4.4.1 MRI_Format Object

```
// See section 7.4.1 on page 188
public class MRI_Format extends Format_Impl
{
    static {
      declareRWKey("DimensionsK");
      declareROKey("PrimitivesK");
    }
}
```

#### 8.4.4.2 EfficiencyMeasure Object

```
// See section 7.4.2 on page 189
abstract public class EfficiencyMeasure extends SimplePREMOObject
{
   abstract public ComparisonRes compare(MRI_Device alternative);
}
```

### 8.4.5 Processing Devices for Media Data

#### 8.4.5.1 MRI_Device Object

```
// See section 7.4.3 on page 190
public interface MRI_Device
  extends VirtualDevice, java.rmi.Remote{}
```

#### 8.4.5.2 Modeller Object

```
// See section 7.4.4 on page 190
public interface Modeller
  extends MRI_Device, java.rmi.Remote{}
```

#### 8.4.5.3 Renderer Object

```
// See section 7.4.5 on page 191
public interface Renderer
  extends MRI_Device, java.rmi.Remote{}
```

#### 8.4.5.4 MediaEngine Object

```
// See section 7.4.6 on page 192
public interface MediaEngine
  extends Renderer, Modeller, java.rmi.Remote{}
```

### 8.4.6 Scene Object

```
// See section 7.6 on page 195
public interface Scene
  extends VirtualDevice, java.rmi.Remote
{
   public void create(Name structName, Object structureType)
      throws AlreadyExists, InvalidType;
   public void attachRead(Name structName, int portId)
      throws NoStructure, MultiplyDefined, BadPort, AccessFailure;
   public void attachWrite(Name structName, int portId)
      throws NoStructure, MultiplyDefined, BadPort, AccessFailure;
   public SceneObjectState inquireStatus(Name structName);
   public void transfer(int portId)
      throws BadPort, NotAttached;
   public void detach(int portId)
      throws BadPort;
   public void delete(Name structName)
      throws NoStructure, Locked;
}
```

### 8.4.7 Objects for Supporting Interaction

#### 8.4.7.1 InputDevice Object

```
// See section 7.5.1 on page 193
public interface InputDevice
  extends Modeller, java.rmi.Remote
{
   public Primitive request();
}
```

#### 8.4.7.2 Router Object

```
// See section 7.5.2 on page 194
public interface Router
  extends MRI_Device, Controller, java.rmi.Remote
{
   public void addConnection(String state, int inputPortId, int
outputPortId)
   throws BadPort, BadState, AlreadyConnected;
public void dropConnection(String state, int outputPortId)
   throws BadPort, BadState;
public Links[] inquireConnections(String state)
   throws BadState;

   public class Links {
      int portA;
      int portB;
   };
}
```

### 8.4.8 Coordinator Object

```
// See section 7.7 on page 199
public interface Coordinator
{
   public interface DeviceInfo{
      public Renderer renderer;
      public int inPortId;
   }
   public void addDevice(Renderer_Impl renderer, int inPortId)
      throws BadPort;
   public void dropDevice(Renderer_Impl renderer);
   public DeviceInfo[]  inquireDevice();
}
```

# Appendix A

## Selected Implementation Issues

This appendix provides additional insight into aspects of the prototype implementation of PREMO which has served us throughout this book. It is not our intention to give a fully detailed overview of the implementation; instead, a few of the non–trivial implementation issues will be highlighted. Our purpose is to help the reader in understanding what goes on "behind the scenes" in a PREMO implementation, thereby gaining a better understanding of the general problems involved in implementing a distributed multimedia environment. The problems we will describe, and the solutions that have been adopted, reflect the chosen environment (i.e., Java and Java RMI) as well as our own software engineering abilities. Consequently, some of the issues listed here might become non–issues if other programming environments are used.

## A.1 The PREMO Environment

### A.1.1 Activity of Objects

Java provides threads as a means to create active objects within a JVM. This feature of Java was one of the decisive factors in choosing Java as an implementation platform because the PREMO model requires the availability of active objects. There is, however, a subtle issue concerning Java threads which is worth mentioning here.

The Java specification does *not* define whether the Java threads scheduler is preemptive or not.[1)] Thread scheduling is done by the platform. Furthermore, thread priority schemes vary from platform to platform and it can be quite difficult to map from one priority scheme to the other. The result of these platform dependencies (which are admittedly difficult to avoid) is a set of threading environments in which it is difficult to ensure equivalent thread behavior. Although not explicitly stated in the Standard, the synchronization model in PREMO only works when active objects get a fair share of the processor which results in an effective emulation of concurrency. At the time of writing, the standard JDK environment running on Windows–NT or W'95, on MacOS8 or higher, as well as on Sun workstations running SunOS 5.6 or higher, use preemptive threads ("native" threads, in the terminology of JDK). However, on SunOS 5.5 for example, the only thread environment available for JDK applications is the so–called

1) There is confusion in the literature about Java in this respect: what we mean by preemptive scheduling is that each thread gets a time slice for execution in some fair manner, *regardless* of what other threads are doing. In contrast, in a non–preemptive thread environment, a compute intensive thread which does no explicit actions which might lead to the suspension of the thread can monopolise the processor, without giving other threads a chance to run.

"green" threads, which is *not* preemptive. In other words, the PREMO implementation is not guaranteed to run properly on SunOS 5.5 or other systems without a preemptive scheduler although judicious application of the yield() statement seems to prevent most problems.

### A.1.2 Top Level of the PREMO Hierarchy

As was formally described in section 5.3 on page 64, a simple PREMO object, should also be a subtype of `PREMOObject`. This is shown in the specification of `SimplePRE-MOObject` below (see also page 65):

```
package premo.std.part2;
public abstract class SimplePREMOObject
   implements PREMOObject, java.io.Serializable {
}
```

Strict adherence to the PREMO object hierarchy in the Java implementation leads to conflicts with RMI. The problem is that subtypes of `SimplePREMOObject` appear in the argument lists and/or the return values of object services, and the idea is that *copies* of such objects are used (i.e., these objects are *not* used for remote services). The current Java RMI implementation uses object serialization to create the copies. These serialized versions are sent through the communication network. However, if the object to be serialized also implements the `java.rmi.Remote` interface, this constitutes a conflict for the current RMI runtime ("Should the object be serialized, or is this an object with a remote stub?") which leads to a runtime exception. On the other hand, if the interface definition above was used to define simple PREMO objects, all simple PREMO objects would also implement `PREMOObject` and, by virtue of inheritance, `java.rmi.Remote`. As a consequence, the reference to `PREMOObject` has to be commented-out in the real implementation (of course, `SimplePREMOObject` still implements, through its own methods, all methods defined in `PREMOObject`).

### A.1.3 Operation Request Modes

Most of the object model, described in Part 1 (see Chapter 3), can be implemented in Java (or indeed in any decent object–oriented environment) without significant problems. The concepts of objects, non–object data, inheritance, etc., are fairly standard. The only problem an implementor might encounter is that PREMO objects might be active, i.e., they may have their own thread of control. Fortunately, threads form an integral part of Java (which is one of the reasons for choosing Java in the first place!), which made our task much easier. If this weren't the case, a smooth integration of threads into the environment might represent a significant amount of work.

Operations in active PREMO objects may also have different operation request modes (see Section 3.8 on page 45). The facilities described in PREMO represent a more "method-oriented" approach to thread synchronization than the basic facilities of Java, which relies on critical sections and object blocking. It is therefore necessary to provide tools to implement these various request modes. There are two main design goals for the PREMO operation request modes which have been provided. These are:

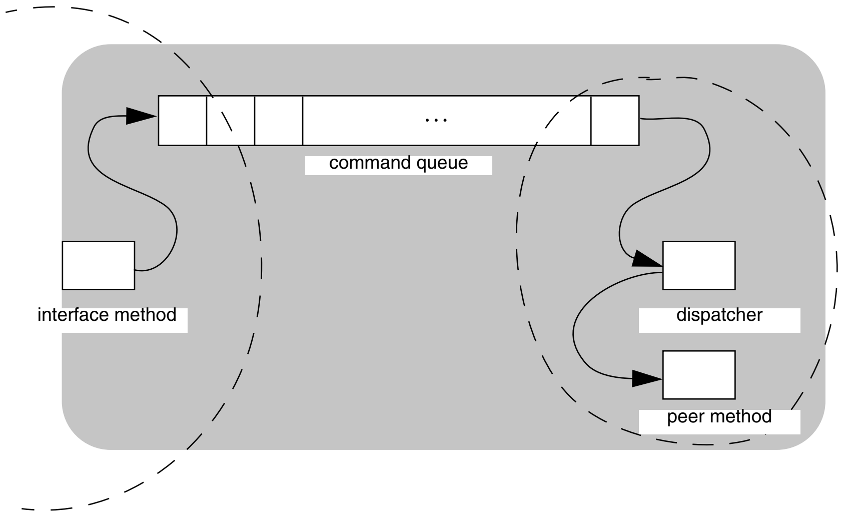


Figure A-1 - Operation Request Implementation

1. Calls to active object operations, i.e., invocation of the object's methods, might be synchronous, asynchronous, or sampled. This should be implemented within the "callee" and there should be no syntactic difference for the caller regardless of the mode being used.
2. The active object should have the means to decide at which stage a certain operation is really performed, and which operation is to be delayed. As an example, we saw in Part 2 that the main processing loop of a synchronization object is "atomic", i.e., no external operation request should be accepted while the object is within this loop, and all callers should be suspended.

Both aspects suggest an implementation scheme depicted on Figure . The gray area on the figure represents an active object; the dotted curve show thread "boundaries". The thread on the left belongs to the caller of an operation, whereas the thread on the right side represents the "real" working thread of the object.

The "interface method" depicted on the left side of the object is the method visible to the caller via the official PREMO interface. What the method does, instead of implementing the real operation, is as follows:

1. An instance of a special class, a "call structure", is created; this class contains the arguments to the call and a reference to the operation to be invoked.
2. If the call is defined to be synchronous or asynchronous, the reference to the class instance is put into a queue, the "command queue". A sampled method requires more care: instead of putting the reference to the call structure instance in the queue, the latter should be inspected first to see if an element referring to the same opera-

tion is already present in the queue or not. If so, the argument values in the call structure should be replaced, instead of putting the new instance into the queue; if not, the reference to the new call structure is put into the queue, just like the asynchronous or synchronous cases.

3. Depending on the specification of the interface method, the caller is either suspended until the call is performed (synchronous case), or the interface method simply returns directly after placing a request for the call on a queue for dispatch (no wait), thereby implementing an asynchronous or a sampled call.

The "callee" side, the real working thread of the object, runs in a loop within a special dispatcher routine, regularly consulting the command queue to see if a new command is available. If yes, the reference to the call structure is retrieved, the reference to the operation is accessed, and the operation doing the real work (the "peer" method) is invoked using the arguments in the call structure.

Although the scheme is simple, some details in the description above merit some more explanation:

- *What is a "reference to an operation"?* Java introduces a class called `Method`, as part of its `java.lang.reflect` package. This object does exactly what we need: one can retrieve it based on the method's signature and the `Class` instance of the containing object, and one can call the `invoke` method on it which invokes the operation corresponding to the `Method` in the containing object. This means that the call structure contains simply a `Method` object, which can be used by the dispatcher to access the peer method. Note that if this class type were not present in Java, implementation of a general dispatch mechanism would be very troublesome indeed!
- *How does synchronization and suspension work in the synchronous case?* This is done by issuing a Java `Object`'s `wait` call *on the call structure instance which is being transmitted in the queue*. When the peer method has completed its work, the dispatcher issues a `notify` on the same object, which releases the caller. Using the `wait` – `notify` pair (which is standard in Java) also allows the dispatcher to fill in return data into the call structure before issuing `notify`; this return value can be retrieved and, ultimately, returned to the caller.
- *What happens to exceptions?* Synchronous operations might also throw exceptions, which should be returned to the caller. Fortunately, the Java mechanism is well prepared for this. Indeed, if an operation is invoked through `Method.invoke`, and the operation throws an exception, `Method.invoke` itself throws a special exception called `InvocationTargetException` whose attribute refers to the original exception. This means that the dispatcher can retrieve this information and send it back via the call structure, instead of providing return values.
- *What is the command queue?* This is a simple wrapper around a `Vector` object which provides synchronized access to its operations, such as putting a new element into the queue, retrieving the head of the queue, and allowing the caller to examine the contents of the queue (for example, to implement the sampled operation). The only reason a wrapper is necessary is to ensure mutually exclusive access to the

contents of the queue. This can be achieved with standard Java `synchronized` statements.

All these facilities are embodied in a separate object called `OperationRequest`, which is part of the `premo.impl.part1` package. This class offers the following operations for the "caller" side:

```
protected Object  callSync(String methodName, Object[] args)
protected void    callAsync(String methodName, Object[] args)
protected void    callSampled(String methodName, Object[] args)
```

and for the "callee" side:

```
protected Call nextCall()
protected Call nextCall(final Method[] methods)
protected Call nextCallNoWait()
protected Call nextCallNoWait(final Method[] methods)
```

The inner class, `Call`, is the call structure referred to in the description. The `nextCall` and `nextCallNoWait` methods without arguments result in an unconditional call using the head of the command queue. If a list of methods is also given as an argument, the first queue element referring to one of those methods is used.

The implementation of `EnhancedPREMOObject`, or `EnhancedPREMOObject_Impl` is a subtype of `OperationRequest`; this means that all PREMO objects have access to these operations to implement their interface methods or their own dispatcher[1]. In a specific case, the interface methods are very simple; for example, the `jump` operation of the synchronizable object (see page 101) can simply be:

```
public void jump(Number position)
{
   callSync("jumpPeer", new Object[] { position });
}
```

Of course, the real work must be done in the `jumpPeer` operation, which is invoked by the internal dispatcher.

Depending on the semantics of the object, the dispatcher may be very simple, too:

```
while(true) {
   nextCall();
```

But, most of the time, the dispatcher is more complex because it has to take into account the internal state of the object. Finally, some PREMO objects may not use these facilities at all because a simpler implementation scheme is possible.

### A.1.4 Distribution and the Creation of PREMO Objects

Each PREMO application is supposed to start by calling

```
PREMORuntime.init()
```

[1] Note that this inheritance relationship was omitted on page 69 to avoid unnecessary confusion at that point.

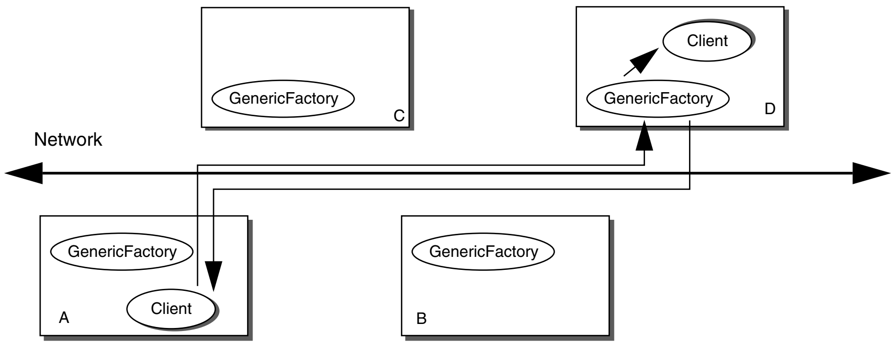


Figure A-2 - Factories and Factory Finders

This static method will initialize a number of system–wide variables which are used by various PREMO implementation objects such as, for example, the internet address of the local host and the name of the machine. More importantly, it will initialize the RMI environment for the local JVM (for example, by setting the correct security manager for RMI). Furthermore, a single local instance of both a generic factory and of a generic factory finder object are created; the naming services of Java RMI are used to "export" the name of the local factory instance (using an agreed upon name, which is simply `"GenericFactory")`. The `PREMOUtil` object also has static variables which refer to these local factory finder and factory instances.

This setting enables the implementation structure shown on Figure , (see also section 5.7.3). A client can retrieve the reference for the local factory finder (using the static variable defined for `PREMOUtil`); this factory finder can be used to locate a reference (an RMI stub) to the remote factory object. Finally, the client can instruct the remote factory object to create a new PREMO object instance and return its stub. The internal implementation of a factory object is a standard sequence of Java RMI calls.

Our current implementation has two restrictions, however:

- The whole structure relies on the fact that only *one* JVM is running on one machine. The reason is that the implementation needs an unequivocal identification of a JVM, for example to set up connections among virtual devices (see section 6.6.3). However, whereas identification of a machine (through the standard `java.net.InetAddress` class) is relatively straightforward in Java, it is much less simple to identify two JVM's running on the same machine, hence sharing an IP address. Furthermore, Java sockets, which are used in our implementation, are bound to IP addresses, too.
- The current factory implementation is limited to creating PREMO object instances on its own JVM, although this restriction does not appear in PREMO.

Although a more professional implementation of PREMO would have to deal with these restrictions, too, neither of them is a terrible setback. Although, for efficiency reasons, one might think of starting up several JVM's on the same machine, a proper implementation of Java with native threads should make such optimization unnecessary.

An unexpected problem did occur, however, when using RMI: If an object instance is "exported" through RMI, which means, to be very precise, that the

```
UnicastRemoteObject.exportObject(newObj);
```

is used to turn the object into a remote service object, it is difficult to get a handle for the "real" object reference of the object and *not* a reference to its stub. Whereas this reference is useless if the object is indeed remote, it is sometimes necessary to have access to the real object if both the caller and the callee are on the same JVM. The reason is that the implementation might need some "hidden" methods which aren't meant for export as remote methods.

The following example illustrates the problem. The `VirtualDevice_Impl` object needs a method of the form:

```
setOutputStream(int portId, OutputStream stream);
```

to record, within its data structures, `stream` as the output channel for the specific port. Obviously, there is no reason why this method would be defined as an RMI call. On the other hand, the objects which set up connections receive references to the device from the PREMO application. What the objects receive is therefore a *stub* for the real implementation object. This stub cannot be used to access `setOutputStream`; consequently, the "real" object reference should be accessed. However, there is no standard way in the current Java RMI mechanism to access to the real object reference, even if the application is sure that both the callee and the caller are on the same JVM!

Because this problem occurs for a number of implementation dependent methods for the various virtual resource objects of Part 3, each JVM maintains (through some static methods and variables of the `VirtualResource_Impl` object) a hashtable containing the references of virtual resource objects. A simple naming mechanism is used to uniquely identify a virtual resource. This is currently done with integers. Each virtual resource carries its own name with it, and this name can be inquired by other objects. Finally, this name can be used to get the object reference from the hashtable.

Of course, there is a bootstrap effect: the caller should be able to retrieve the name and the IP location of the object in order to retrieve the real object reference. We have chosen therefore to add a very small set of remote methods to the specification of a virtual resource, which return the data used for a unique identification (essentially, the IP address and the unique name of the resource). No other remote methods are necessary for the implementation.[1)]

## A.2 Specific Part 3 Objects

### A.2.1 Virtual Connection Objects

The difficulties of implementing virtual connection objects are due to the several communication paradigms which have to be combined within the same application. Therefore, the first question to answer is obviously: is it really necessary to combine several

[1)] Note that the standard PREMO property mechanism might have been used to solve the problem. The use of a restricted set of remote methods is only a matter of convenience.

paradigms? It is indeed possible to exclusively use RMI calls to transfer data from among JVM's. Some sort of `PREMOMediaStream` service object which actively transfers data from one port to another could conceivably be used. Because RMI objects and stubs are smoothly integrated with the rest of the Java environment, the use of such objects could make all communication details transparent to the device implementations, too.

This approach was rejected in favor of speed: transferring data through RMI calls is much slower than transferring the same data through sockets. While the difference is not noticeable for the transfer of one or two average sized objects (used in a typical method invocation), it is prohibitive where media data is concerned. Hence, the more complex approach, described in section 6.6.3, was required.

This scheme differentiates between two cases: when both devices are on the same JVM, and when they run on different JVM's. The `VirtualConnection_Impl` class has the task of differentiating between the two cases and, possibly, performing some additional checks such as checking the media formats on the ports which are to be connected. To ensure a better modularization of the code, the communication–specific code has been placed in a separate implementation class called `Connector_Impl`. There is a single instance of this class running per JVM, which is also "exported" through the RMI. It is the task of the `PREMOUtil` class to start up this one instance. It is through methods of this class that the connections are actually made. These methods are invoked by the `VirtualConnection_Impl` class.

#### A.2.1.1 Devices on the Same JVM: Piped Streams

The "easy" half of the `Connector_Impl` object is to set up piped streams between two virtual devices sharing the same JVM. The definition of these pipes is a standard technique in Java for setting up a communication channel between two threads. The only difficulty is the problem already mentioned in section A.1.4: the `Connector_Impl` object needs to have direct access to the virtual device instance, and not just its stub. However, the general naming mechanism of virtual resources and the corresponding extra remote methods solve this problem as well.

#### A.2.1.2 Devices on Different JVM's: Sockets

If the devices are on different JVM's, the task of setting up communication between the two ports involves two steps:

1. Set up a dedicated socket pair for this communication channel.
2. Connect the ports to the Java streams associated with the socket.

Obviously, the second step can only be done on the JVM where the target port resides, which suggests that setting up the connection is done "in cooperation" by the two `Connector_Impl` object instances residing on their respective JVM's.

We will not dive into the details of the Java socket mechanism here. There are excellent overviews of this subject and we expect the reader to be familiar with sockets. What is important to emphasize is that each JVM has to run a separate thread whose only purpose is to "accept" socket creation requests. This is how new socket pairs are created. Each `Connector_Impl` instance spawns such a thread which we will refer to as the `Watcher`.

Setting up a new connection involves a little communication protocol between the two `Connector_Impl` instances residing on the two JVM's. As an agreement, the protocol is always initiated by the `Connector_Impl` object residing at the "source" of the stream. When initiating a connection, the object on the source side tries to get a socket, by attempting to create a `java.net.Socket` object, using the sink's internet address and a fixed socket port number. If this succeeds, a kind of "handshake" takes place between the `Connector_Impl` of the source and the `Watcher` on the sink side. The source can then take the output part of the socket and use this stream to set the output port of the relevant device port.

The problem is identifying the corresponding socket on the sink side. One should not forget that, theoretically, two connection requests can simultaneously arrive at the same sink JVM, so there is a potential race condition in uniquely identifying the newly created socket. It is necessary to uniquely identify the socket pair on the sink side even if several socket creation requests interfere with one another.

The way of uniquely identifying the socket is based on the IP number and the socket port (not the virtual device port!) number. If the sink side `Watcher` defines a server socket through:

```
ServerSocket master;
master = new ServerSocket(AgreedPortNumber, SinkInetAddress);
```

and contains a routine including:

```
Socket s = master.accept();
// got a request from a source
String IP1 = (s.getInetAddress()).getHostAddress();
int    ID1 = s.getPort();
```

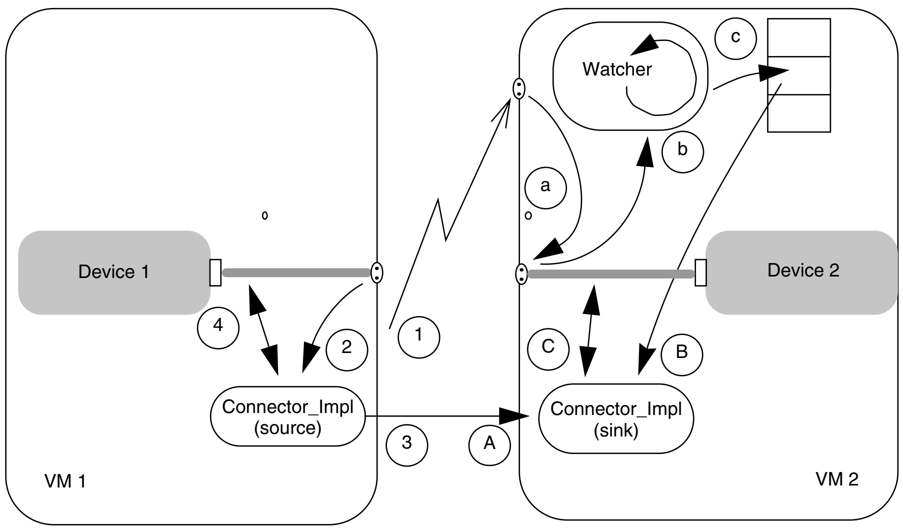
while, at the same time, the source side `Connector_Impl` does:

```
Socket s = new Socket(AgreedPortNumber,SinkInetAddress);
// accepted request
String IP2 = (s.getLocalAddress()).getHostAddress();
int    ID2 = s.getLocalPort();
```

then the values of the tuples `<IP1,ID1>` and `<IP2,ID2>` will be identical.

Based on this observation, the following happens (see also Figure ): When the `Watcher` gets a request, it stores the socket reference in a hashtable, using the `<IP1,ID1>` tuple as a key. It then continues processing, possibly getting new requests. The source side `Connector_Impl` retrieves its own `<IP2,ID2>` tuple; this will be sent *through a remote method call*, to the `Connector_Impl` on the sink side.The latter can retrieve the socket reference from the hashtable (it may have to try several times, because the `Watcher` might not have been scheduled to put the reference into the hashtable yet). Once the socket reference is found, the corresponding `InputStream` is retrieved and attached to the virtual device, thus establishing a connection.

The choice of "source" and "sink" is dependent on the choice of the output and input ports connected to the media stream. Obviously, each Java VM can play the role of both a "source" side and a "sink" side, depending on the PREMO application.



1. *Source creates a Socket instance connecting to a server socket.*
2. *<IP2,ID2> is extracted.*
3. *<IP2,ID2> is transferred to sink.*
4. *Socket and port is connected with a Java stream.*

a. *Watcher accepts a socket request.*
b. *Watcher extracts <IP1,ID1>.*
c. *Watcher stores the socket reference in a hashtable.*

A. *Sink receives <IP2,ID2>.*
B. *Sink retrieves socket reference from the hashtable.*
C. *Socket and port is connected with a Java stream.*

Figure A-3 - Setting up a Socket Pair for Media Streams

### A.2.1.3 Multicast Connections

Using Java streams for connection makes it relatively straightforward to implement a “fan–out” type multicast connection. `FanOutStream` is defined in Part3 as a subtype of the standard `OutputStream` of Java. This fan–out stream offers the capability of attaching to and detaching from other output streams. The class overrides the standard `write` operations to copy each byte to all attached streams.

In the simplest case, such a `FanOutStream` object instance can be plugged into the output port of the master device and, from that point on, attaching a new connection to the device means, eventually, to attach one more output stream to the fan out stream (of course, the “real” output stream has to be created following the same procedures as for a single connection). A more optimal use of a `FanOutStream` is conceivable: by analysing the network requirements (through the internet number of each virtual device in a network), `FanOutStream` instances could be put “closer” to the consumer, to avoid transferring multiple copies of the same data on the same route. An easy case is if two slave devices share the same JVM; a `FanOutStream` could be put onto this target JVM instead of the one running the master. We have not implemented such optimizations in our prototype, but it could be added without too much work.

The difficulty lies in implementing a "fan–in" multicast connection. Unfortunately, all standard Java IO operations boil down to reading one single byte through the standard `read` operation of `InputStream` so it seems impossible to provide some sort of `FanInStream` in full generality which would also ensure the consistency of larger data packets. Consequently, our prototype implementation does not provide a fan–in connection.[1)]

[1)] Note that if the RMI mechanism was used for the virtual connection, implementation of a fan–in type connection would not represent a real problem. Data would be transferred through method arguments, i.e., the consistency of data would be automatically preserved. Unfortunately, the speed of the current RMI implementation makes this approach impractical.

# References

[1] P. Ackermann. *Developing Object-Oriented Multimedia Software - Based on the MET++ Application Framework*, dpunkt Verlag, Heidelberg, 1996.

[2] W. Appelt and A. Scheller. HyperODA: Going Beyond Traditional Document Structures. *Computer Standards & Interfaces*, 17(1):13–21, 1995.

[3] F. Arbab, I. Herman, and G.J. Reynolds. An Object Model for Multimedia Programming. *Computer Graphics Forum*, 12(3):C101–C113, 1993.

[4] F. Arbab: "The IWIM model for coordination of concurrent activities". In: Coordination Languages and Models, Springer Verlag, Lecture Notes in Computer Science, vol. 1061 series, Berlin - Heidelberg - New York, pp. 34-56, 1996.

[5] D.B. Arnold and D.A. Duce. *ISO Standards for Computer Graphics: The First Generation.* Butterworth, 1990.

[6] M. Awad and J. Ziegler. A Practical Approach to the Design of Concurrency in Object–Oriented Systems. *Software — Practice and Experience*, 27(9):1013–1034, 1997.

[7] J. Barnes. *Programming in Ada'95*. Addison–Wesley, 1996.

[8] D.R. Begault. *3D Sound for Virtual Reality and Multimedia*. Academic Press, 1994.

[9] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, K. Nygaard. *Simula Begin*. AUERBACH Publishers Inc., 1973

[10] G. Blakowski and R. Steinmetz. A Media Synchronization Survey: Reference Model, Specification, and Case Studies. *IEEE Journal on Selected Areas in Communications*, 14(1):5–35, 1996.

[11] D.G. Bobrow, L.G. Demichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, D.A. Moon. Common Lisp Object System Specification. *Lisp and Symbolic Computation*, 1(3/4), 1989.

[12] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1986.

[13] G. Booch. *Object–Oriented Analysis and Design with Applications (Second edition)*. Prentice–Hall, 1997.

[14] D. Brookshire Conner and A. van Dam. Sharing Between Graphical Objects Using Delegation. In C. Laffra, E.H. Blake, V. de Mey, and X. Pintado, editors, *Object–Oriented Programming for Graphics*, Springer–Verlag, Focus on Computer Graphics Series, 1995.

[15] N. Carriero and D. Gelernter: "Linda in Context". In: Communication of the ACM, 32, pp. 444-458, 1989.

[16] L. Chamberland. *FORTRAN 90: A Reference Guide*. Prentice Hall, 1996.

[17] F. Colaïtis and F. Bertrand. The MHEG Standard: Principles and Examples of Applications. In W. Herzner and F. Kappe, editors, *Multimedia/Hypermedia in Open Distributed Environments*, Springer–Verlag, 1994.

[18] R.B. Dannenberg, T. Neuendorffer, J. Newcomer, D. Rubine, and D. Anderson. Tactus: Toolkit-level Support for Synchronized Interactive Multimedia. *Multimedia Systems Journal*, 1(2):77–86, 1993.

[19] D.A. Duce, D.J. Duke, P.J.W. ten Hagen, I. Herman, and G.J. Reynolds. Formal Methods in the Development of PREMO. *Computer Standards & Interfaces*, 17:491–509, 1995.

[20] D.J. Duke, D.A. Duce, I. Herman and G. Faconti. *Specifying the PREMO synchronization objects*. Technical report 02/97-R048, European Research Consortium for Informatics and Mathematics (ERCIM), 1997.
URL ftp://ftp.inria.fr/associations/ERCIM/research_reports/pdf/0297R048.pdf

[21] D.J. Duke and I. Herman. Programming Paradigms in an Object-Oriented Multimedia Standard. In P. Slusallek and F. Arbab, editors, *Proc. of the Eurographics Workshop on Programming Paradigms in Computer Graphics*, Eurographics Publications Series, 1997.

[22] D.J. Duke, I. Herman, T. Rist, and M. Wilson. Relating the primitive hierarchy of the PREMO standard to the Standard Reference Model for Intelligent Multimedia Presentation Systems. *Computer Standards & Interfaces*, 20, 1998.

[23] D.A. Duce, D.J. Duke, I. Herman and G. Faconti. The Changing Face of Standardization: A Place for Formal Methods? *Formal Aspects of Computing*, 11, 1999, in press.

[24] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language, Version 1. Technical Report, The University of Queensland, No. 91-1, 1991.

[25] R. Duke, G. Rose, and G. Smith. Object–Z: A Specification Language Advocated for the Description of Standards. *Computer Standards & Interfaces*, 17(5-6):511–534, 1995.

[26] G. Faconti and M. Massink. Investigating the Behaviour of PREMO Sychronization Objects. In *Proceedings of the 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, Springer–Verlag, 1997.

[27] B.N. Freeman–Benson and A. Borning. Integrating constraints with an object–oriented language. In I. Lehrmann Madsen, editor, *Proceedings of the ECOOP'92 European Conference on Object–Oriented Programming,* Springer Verlag, Lecture Notes in Computer Science 615, 1992.

[28] D. Flanagan. *Java in a Nutshell (Second edition)*. O'Reilly, 1997.

[29] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison–Wesley, 1990

[30] M. Fowler, K. Scott. UML Distilled: Applying the Standard Object Modeling Language. Addison Wesley, 1997.

[31] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design patterns: elements of reusable object-oriented software*, Addison Wesley, 1995.

[32] A. Goldberg and D. Robson. *Smalltalk–80: The Language*. Addison–Wesley, 1989.

[33] S.J. Gibbs, L. Dami, and D.C. Tsichritzis. *An Object Oriented Framework for Multimedia Composition and Synchronisation*. In L. Kjelldahl, editor, *Multimedia (Systems, Interaction and Applications)*, Springer–Verlag, 1992.

[34] S.J. Gibbs and D.C. Tsichritzis. *Multimedia Programming*. Addison–Wesley, 1995.

[35] A. Goldberg, D. Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[36] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison–Wesley, 1996.

[37] S. Green. *Parallel Processing for Computer Graphics*. Pitman, 1991.

[38] J. Hartman and J. Wernecke. *The VRML 2.0 Handbook*. Addison–Wesley, 1996.

[39] P. Heller, S. Roberts, P. Seymour and T. McGinn. *Java 1.1 Developer's Handbook*. Sybex, 1997.

[40] I. Herman, G.J. Reynolds, and J. Van Loo. PREMO: An emerging standard for multimedia. Part I: Overview and Framework. In: *IEEE MultiMedia*, 3(3):83–89, 1996.

[41] I. Herman, N. Correia, D.A. Duce, D.J. Duke, G.J. Reynolds, and J. Van Loo. A Standard Model for Multimedia Synchronization: PREMO Synchronization Objects. *Multimedia Systems*, 6, 1997.

[42] I. Herman, G.J. Reynolds, and J. Davy. MADE: A Multimedia Application development environment. In L.A. Belady, editor, *Proc. of the IEEE International Conference on Multimedia Computing and Systems, Boston*, IEEE CS Press, 1994.

[43] C.A.R. Hoare. *Communicating Sequential Processes*. Addison–Wesley, 1985.

[44] T.L.J. Howard, W.T. Hewitt, R.J. Hubbold, and K.M. Wyrwas. *A Practical Introduction to PHIGS and PHIGS PLUS*. Addison Wesley, 1991.

[45] IMA, *Multimedia System Services*, Interactive Multimedia Association, September 1994, ftp://ima.org/pub/mss/.

[46] Information Processing Systems — Open Systems Interconnections — LOTOS (Formal Description Technique based on the temporal ordering of observational behaviour). International Standardization Organization, ISO/IS 8807, 1989.

[47] Information Processing Systems — Computer Graphics — Computer Graphics Reference Model (CGRM), International Organisation for Standardization, ISO/IEC IS 11072, 1992.

[48] Information technology — Computer graphics and image processing — Graphical Kernel System (GKS), Part 1: Functional description. International Organization for Standardization, ISO/IEC 7942–1, 1994

[49] Information Technology — Computer Graphics — Programmer's Hierarchical Interactive Graphics System (PHIGS). International Organisation for Standardization, ISO/IEC IS 9592 1997.

[50] Information Technology — Computer Graphics and Image Processing — Presentation Environments for Multimedia Objects (PREMO), Part 1: Fundamentals of PREMO. International Organization for Standardization, ISO/IEC 14478–1, 1998.

[51] Information Technology — Computer Graphics and Image Processing — Presentation Environments for Multimedia Objects (PREMO), Part 2: Foundation Component. International Organization for Standardization, ISO/IEC 14478–2, 1998.

[52] Information Technology— Computer Graphics and Image Processing — Presentation Environments for Multimedia Objects (PREMO), Part 3: Multimedia Systems Services. International Organization for Standardization, ISO/IEC 14478–3, 1998.

[53] Information Technology — Computer Graphics and Image Processing — Presentation Environments for Multimedia Objects (PREMO), Part 4:Modelling, Rendering, and Interaction Component. International Organization for Standardization, ISO/IEC 14478–4, 1998.

[54] Information Technology — Computer Graphics and Image Processing — The Virtual Reality Modeling Language (VRML), Part 1: Functional specification and UTF-8 encoding. International Organization for Standardization, ISO/IEC 14772–1, 1998.

[55] Information Technology — Coding of Moving Pictures and Associated Audio for Digital Storage up to about 1.5 Mbit/s (MPEG). International Organisation for Standardization, ISO/IEC 10744, 1992.

[56] ISO/IEC directives: Procedures for the technical work of ISO/IEC JTC 1 on Information Technology, 1995. See http://www.iso.ch/dire/jtc1/directives.htm

[57] R.S. Kalawsky. *The Science of Virtual Reality and Virtual Environments*. Addison–Wesley, 1994.

[58] B. Kernighan and D. Ritchie. *The C Programming Language*, second edition. Prentice Hall, 1989.

[59] J. F. Koegel Buford, editor. *Multimedia Systems*. Addison–Wesley, 1994.

[60] J. F. Koegel Buford. Architecture and Issues for Distributed Multimedia Systems. In J. F. Koegel Buford, editor. *Multimedia Systems*. Addison–Wesley, 1994.

[61] C. Laffra, E.H. Blake, V. de Mey, and X. Pintado, editors. *Object–Oriented Programming for Graphics*. Springer–Verlag, Focus on Computer Graphics Series, 1995.

[62] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proc. OOPSLA'86*, ACM Press,1986.

[63] A. Lie and N. Correia. Cineloop Synchronization in the MADE Environment., in: *Proceedings of the IS&T/SPIE Symposium on Electronic Imaging, Conference on Multimedia Computing and Networking*, San Jose, 1995.

[64] B. MacIntyre and S. Feiner. Future Multimedia User Interfaces. *Multimedia Systems*, 4(5):250–268, 1996.

[65] V. de Mey and S. Gibbs. A Multimedia Component Kit. In P.V.Rangan, editor, *Proceedings of the First ACM International Conference on Multimedia (MM93)*, ACM Press, 1993.

[66] B. Meyer. *Eiffel: The Language*. Prentice–Hall, 1990.

[67] B. Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

[68] R. Newcomb, N.A. Kipp, and V.T. Newcomb. The "HyTime" — Hypermedia/ Time–based Document Structuring Language. *Communication of the ACM*, 34(11):67–83, 1991.

[69] Object Management Group. See http://www.omg.org/

[70] R. Orfali and D. Harkey. *Client/Server Programming with JAVA and CORBA*. Wiley Computer Publishing, 1997.

[71] R. Otte, P. Patrick, and M. Roy. *Understanding CORBA, The Common Object Request Broker Architecture*. Prentice–Hall, 1996.

[72] G.J. Reynolds, D.A. Duce, and D.J. Duke. Report of the ISO/IEC JTC1/SC24 Special Rapporteur Group on Formal Description Techniques. Doc. No. ISO/IEC JTC1/SC24 N1152, 1994.

[73] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modeling and Design*. Prentice–Hall, 1991.

[74] W. Schroeder, K. Martin, B. Lorensen. *The Visualization Toolkit*, second edition. Prentice Hall, 1998.

[75] P. Slusallek and H.–P. Seidel. Vision: An Architecture for Global Illumination Calculations. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):77–96, 1995.

[76] M.A. Srinivasan and C. Basdogan, Hapics in virtual environments: taxonomy, research status, and challenges, *Computers & Graphics*, 21(4), Pergamon, 1997.

[77] J. Smith. *X: A Guide for Users*. Prentice Hall, 1994.

[78] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice–Hall, 1992.

[79] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, 1990.

[80] A.S. Tanenbaum. *Modern Operating Systems*. Prentice–Hall, 1992.

[81] H. Tokuda. Operating System Support for Continuous Media Applications. In J.F. Koegel Buford, editor. *Multimedia Systems*. Addison–Wesley, 1994.

[82] B.J. Torby. *FORTRAN'77 for Engineers*. Prentice Hall, 1990.

[83] D. Ungar, R.B. Smith. SELF: The Power of Simplicity. *LISP and Symbolic Computation*, 4(3), Kluwer, 1991.

[84] C. Upson, T. Faulhaber Jr., D. Kamins, et al. The Application Visualization System: A Computational Environment for Scientific Visualization. IEEE Computer Graphics and Applications, 9(4), 1989.

[85] M. Vazirgiannis and T. Sellis. Event and Action Representation and Composition for Multimedia Application Scenario Modelling. In E.Möller, and H. Pusch, editors, *Interactive Distributed Multimedia Systems and Services, Proceedings of the European Workshop IDMS'96*, Springer–Verlag, 1996.

[86] A. Vogel and K. Duddy. *Java Programming with CORBA*. IEEE Press, 1997.

[87] A. Watters, G. van Rossum, J.C. Ahlstrom. *Internet Programming with Python*. M&T Books, 1996.

[88] P. Wegner, S.B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In S. Gjessing and K. Nygaard, editors, *ECOOP'88: European Conference on Object-Oriented Programming*. Volume 322 of Lecture Notes in Computer Science, Springer, 1988.

[89] J. Wernecke, *The Inventor Mentor*, Addison Wesley, 1994.

[90] R. Wirfs–Brock and R. Johnson. Surveying Current Research in Object–Oriented Design. *Communications of the ACM*, 33(9):104–123, 1990.

[91] A. Wollrath, J. Waldo, and R. Riggs. Java-Centric Distributed Computing. *IEEE Micro*, 17(3):44–53.

[92] P. Wißkirchen. *Object–Oriented Graphics*. Springer–Verlag, 1990.

[93] M. Woo, J. Neider, T. Davis. *OpenGL Programming Guide*, second edition. Addison Wesley, 1997.

[94] W3C PNG (Portable Network Graphics) Specification. Public document, available at http://www.w3.org/TR/REC-png

[95] W3C SMIL Draft Specification. Public document, available at http://www.w3.org/TR/WD-smil, 1998.

# Index